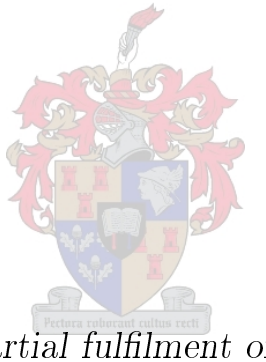
Fast Array Analysis Using Parallelised Domain Decomposition Methods

by

Tameez Ebrahim



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University

Supervisor: Dr. D. Ludick

September 2020

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 2020/09/03 .....

Abstract

Fast Array Analysis Using Parallelised Domain Decomposition Methods

T. Ebrahim

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MScEng(E&E)

December 2020

The requirements for antennae increase daily and as such, antenna arrays are used to meet them. These arrays are often large, consisting of hundreds of antennae, and typically require multiple simulations for optimisation. Due to the computational requirements needed for these runs, it is necessary to apply specialised methods in order to reduce runtime while maintaining accuracy. In this thesis, three domain decomposition techniques based on the Method of Moments (MoM), namely, the Characteristic Basis Function Method (CBFM), the Domain Green's Function Method (DGFM) and the Improved Domain Green's Function Method (i-DGFM) will be investigated and implemented. These techniques will then be parallelised using shared and distributed high performance computing methods such as Open Multi Processing (OpenMP) and Message Passing Interface (MPI). In addition, implementation and parallelisation will also be performed for a Graphical Processing Unit (GPU).

Uittreksel

Vinnige Antenna Samestelling Analise met Parallel Gebied Ontbinding Metodes

T. Ebrahim

Departement Elektriese en Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MScIng (E&E)

Desember 2020

Die vereiste vir groot antenna samestellings neem daagliks toe. Hierdie groot samestelling bestaan uit honderde antennas en benodig tipies verskeie simulasies vir optimering doeleindes. Weens the verwerkingsvereistes wat hiervoor benodig word, is dit nodig om spesialis metodes in te span om sodoende the looptyd te verminder, terwyl die akkuraatheid van die oplossing behoue bly. In hierdie tesis word drie gebied ontbindings metodes ondersoek wat almal onderliggend op die Moment Metode (MoM) baseer is, naamlik die Karakteristieke Basis Funksie Metode (KBFM), the Gebied Green's Funksie Metode (GGFM) en die Verbeterde Gebied Green's Funksie Metode (v-GGFM). Hierdie tegnieke word geparalleliseer deur middel van gedeelde en verspreide hoÃ« spoed verwerking tegnieke soos oop multi ververking (OpenMP) en boodskap deel koppelvlak (MPI). Addisioneel word die parallelisering deur middel van 'n Grafiese Verwerker (GPU) ook ondersoek.

Acknowledgements

I would like to express my sincere gratitude to the following people:

- My parents for their support.
- Dr Ludick for all the help and guidance.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

API	Application Programming Interface
CAD	Computer Aided Design
CBFM	Characteristic Basis Function Method
CEM	Computational Electromagnetics
CHPC	Centre for High Performance Computing
CUDA	Compute Unified Device Architecture
DGFM	Domain Green's Function Method
EFIE	Electric Field Integral Equation
GPU	Graphical Processing Unit
HPC	High Performance Computing
i-DGFM	Improved Domain Green's Function Method
MBF	Macro Basis Functions
MoM	Method of Moments
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
PEC	Perfect Electric Conductor
RAM	Random Access Memory
SIMT	Single Instruction Multiple Threads
SKA	Square Kilometre Array
SMP	Shared Memory Programming
SPMD	Single Program Multiple Data
SVD	Singular Value Decomposition
RWG	Rao-Wilton-Glisson

Chapter 1

Introduction

Humanity is advancing in leaps and bounds in the areas of science and technology. One of the leading areas in this advancement is the field of wireless communication, and as the medium, antennas are required to fulfil ever increasing criteria in terms of stability and performance. After a certain point, singular antennas are not able to fulfil the requirements effectively and it is therefore necessary to combine antennas in specific configurations called antenna arrays.

An example of this is the Square Kilometre Array (SKA)[1], which requires extreme sensitivity in order to monitor the sky in great detail. To achieve this, a singular antenna would need to be very large, such as the Arecibo radio telescope [5], which is 305 meters in diameter. Comparatively, the SKA would use multiple smaller dishes, such as shown in Figure 1.1, to achieve an even greater sensitivity.

Choosing an antenna array has its own challenges and while intuition is needed for the initial array design, it quickly becomes apparent that multiple simulations are required to meet design goals. It is unrealistic to expect the effects of these to be calculated manually, either due to sheer size, or irregular geometry. It is therefore necessary to introduce computational methods in order to effectively simulate the design and gauge the performance.

This thesis will therefore continue the work presented in the author's article written for a peer reviewed IEEE conference,

Ebrahim, T. and Ludick, D.J.: A parallelized fast array analysis approach. In: *2020 14th European Conference on Antennas and Propagation (EuCAP)*, pp. 1-3. 2020 [6],

and investigate and implement three computational methods, namely, the Characteristic Basis Function Method (CBFM), the Domain Green's Function Method (DGFM) and the Improved Domain Green's Function Method (i-DGFM). These methods will then be accelerated using high performance computing techniques in order to gauge their scalability.

Figure 1.1: A few of the antennae in the the SKA mid to high frequency 15m dish array [1].

1.1 Antenna simulation using Computational Electromagnetics (CEM)

Antennas operate using electromagnetic waves, which are described by Maxwell's equations which in modern vector form are written as

$$\nabla \times \mathbf{E} = -\frac{\partial}{\partial t}\mathbf{B} \tag{1.1}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial}{\partial t}\mathbf{D} \tag{1.2}$$

$$\nabla \cdot \mathbf{D} = \rho \tag{1.3}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{1.4}$$

with the consecutive relations given as

$$\mathbf{B} = \mu\mathbf{H} \tag{1.5}$$

and

$$\mathbf{D} = \epsilon\mathbf{E}. \tag{1.6}$$

When applying these formulae to real world problems, the solutions become increasingly complex. Computational Electromagnetics (CEM) therefore aims

to calculate approximate solutions to these formulae. Some methods are detailed below.

1.1.1 An overview of full-wave CEM Methods

The core of full-wave methods is the discretisation of an unknown electromagnetic property as mentioned in [7]. Discretisation involves subdividing the geometry into smaller parts, with the method of subdivision being based on the specific CEM solution approach being applied (such as two dimensional triangular patches or three dimensional cubes). In Figure 1.2, a sphere is discretised, or meshed, into triangular patches. Discretisation plays an important role, as the finer the geometry is discretised, the more accurate the solution.

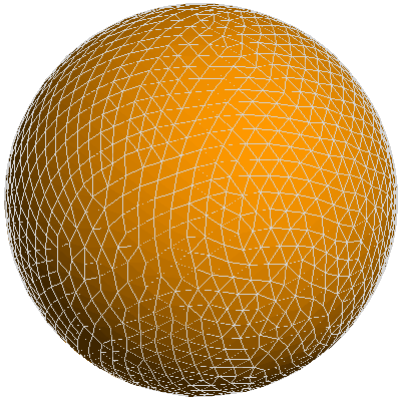


Figure 1.2: A sphere discretised/meshed using triangular patches

Three of the more popular full-wave methods are the Finite Element Method (FEM) [8], the Finite Difference Time Domain (FDTD) method [9], and the Method of Moments (MoM) [4]. The FEM and FDTD are alike in that both aim solve the differential form of Maxwell's' equations while the MoM aims to solve the integral form. The MoM requires only the surface of the geometry to be discretised, as opposed to other approaches, such as the FDTD, that require the surrounding space to be discretised as well [7]. This makes the MoM particularly efficient with respect to the solving of pure antenna structures and is why the MoM is chosen as the method of choice in this work.

1.1.2 A brief overview of the Method of Moments (MoM)

In the MoM, the geometry is replaced with equivalent discretised surface currents [7]. The discretisation varies in the geometry and material properties. The more commonly used discretisation approaches includes the use of segments (for thin wires), or triangular patches, mentioned in [4], for surface

structures. Triangular patches are used because it can conform to nearly any geometrical shape. As mentioned in [7], a matrix equation is derived from the interactions between all the aforementioned discretised segments. These interactions are calculated using a Green function, the free-space Green function in the case of this work. A set of linear equations are then obtained by imposing a set of boundary conditions on the interactions. This results in a fully populated matrix equation,

$$[Z][I] = [V], \tag{1.7}$$

where $[Z]$ is the complex impedance matrix, $[I]$, the surface currents to be solved for and $[V]$, the excitation vector. The dimensions are $N \times N$ for $[Z]$ and $N \times 1$ for $[V]$ and $[I]$, with N the amount of discretised elements, be it segments or patches. $[I]$ is then solved for using linear algebraic methods, the method of choice in this work being the LU-decomposition. The specific formulation and solution of Equation (1.7) will be discussed in detail in Chapter 4.

In the case of an antenna array, one can imagine that N can quickly become large ($[Z]$ scales as $\mathcal{O}(N^2)$). This poses problems in terms of computational speed as well as computer memory limitations.

1.1.2.1 Implications of a large N in terms of speed and memory

While the filling of a large $[Z]$ matrix in (1.7) is a relatively slow process, it is not easily escaped from, even when utilizing more advanced techniques. The main issue in this case is the solving of $[I]$ in (1.7) which when using the LU-decomposition method is of $\mathcal{O}(N^3)$. This increase in N also impacts computer memory as the memory used by $[Z]$ scales as $\mathcal{O}(N^2)$.

With regards to antenna arrays, both these problems can be alleviated using domain decomposition techniques as presented below.

1.1.3 Domain decomposition techniques

Consider the matrix, $[Z]$ in (1.7). A finer description based on a simple square perfect electric conductor (PEC) plate, p, in Figure 1.3 is

$$\begin{bmatrix} Z_{11} & Z_{12} & Z_{13} & Z_{14} \\ Z_{21} & Z_{22} & Z_{23} & Z_{24} \\ Z_{31} & Z_{32} & Z_{33} & Z_{34} \\ Z_{41} & Z_{42} & Z_{43} & Z_{44} \end{bmatrix}, \tag{1.8}$$

where Z_{xy} is the interaction between the discretised current components flowing over the interior edges x and y. Now, in the context of an antenna array, in this case the two square plates p and q in Figure 1.3, $[Z]$ can be represented as

$$\begin{bmatrix} [Z_{pp}] & [Z_{pq}] \\ [Z_{qp}] & [Z_{qq}] \end{bmatrix}, \tag{1.9}$$

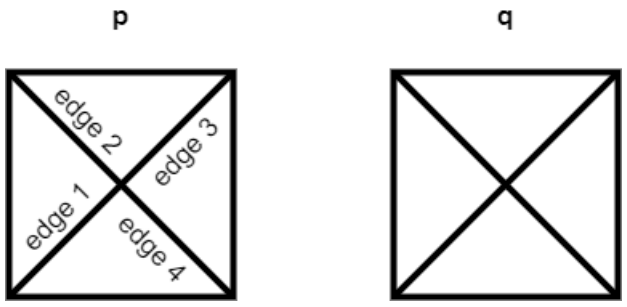


Figure 1.3: Two discretised square plates, p and q.

where $[Z_{ab}]$ is the interaction sub-matrix between plates a and b. It follows that $[Z_{aa}]$ is simply (1.8). Now, considering an arbitrary amount (M) of antennae in an array, the full description of (1.7) becomes

$$\begin{bmatrix} [Z_{11}] & [Z_{12}] & \cdots & [Z_{1M}] \\ [Z_{21}] & [Z_{22}] & \cdots & [Z_{2M}] \\ \vdots & \vdots & \ddots & \vdots \\ [Z_{M1}] & \cdots & \cdots & [Z_{MM}] \end{bmatrix} \begin{bmatrix} [I_1] \\ [I_2] \\ \vdots \\ [I_M] \end{bmatrix} = \begin{bmatrix} [V_1] \\ [V_2] \\ \vdots \\ [V_M] \end{bmatrix}. \tag{1.10}$$

Domain decomposition techniques seek to exploit the block like nature of (1.10), by either reducing or partitioning the problem to improve speed and memory usage.

1.2 Research objectives

The aim of this work is to:

- Investigate and implement three domain decomposition techniques, namely the Characteristic Basis Function Method (CBFM) [10], the Domain Greens Function Method (DGFM) [11] and the Improved Domain Greens Function Method (i-DGFM) [12].
- Apply parallelisation to improve the speed of the aforementioned techniques using MPI [13], OpenMP [14], and GPU acceleration.
- Compare the mentioned techniques in terms of performance, accuracy and scalability.

1.3 Thesis structure

The thesis is structured as follows: In Chapter 2, an investigation into high performance computing and parallel computing techniques related to the nature of domain decomposition methods are presented. In Chapter 3, the author's

computational software, CEMACS, is discussed with regard to implementation and methods of validation. In Chapter 4, a formulation of the Method of Moments (MoM) is presented along with implementation considerations and validation results. In Chapters 5, 6 and 7, the Characteristic Basis Function Method (CBFM), the Domain Green's Function Method and the Improved Domain Green's Function Method (i-DGFM) are formulated, implemented, validated and parallelised respectively. In Chapter 8, brief comparisons between the implemented domain decomposition methods are made. Finally, the thesis is concluded in Chapter 9 with a summary overview and directions for future research.

Chapter 2

High Performance Computing

This chapter presents an overview of high performance computing techniques and how they can be used effectively. A brief motivation will be presented followed by the discussion of the high performance techniques that will be used in this work. This includes the Message Passing Interface (MPI) [13], and Open Multi Processing (OpenMP) [14]. The use of Graphical Processing Units (GPU's) will also be discussed. Specific focus is placed on costs and benefits of these techniques as well as general implementation strategies.

2.1 The current state of computing

As time passes, computing power continues to become both greater and cheaper. As per Moore's law, the transistor density doubles roughly every one and a half to two years [2]. While this remains true, single processor performance has started to peter off as shown in Figure 2.1.

The clock frequency in Figure 2.1 refers to the amount of instructions a processor can perform per second. It is therefore worrying to see the stagnation in growth as the requirement of computing power continues to increase. As mentioned in [2] this stagnation is due to power and energy constraints, as well as the dissipation of heat in the processor. This effect is illustrated in Figure 2.2 below.

Per [2], 130 W is the physical limit for air cooling, requiring the use of large heat sinks and fans. It is therefore recommended in [2] that multiple processors of lower specifications be coupled together rather than a single powerful unit, in order to continue to achieve the required computing power. It is therefore unsurprising that the processor industry has moved in this direction. With the coupling of processors, multiple instructions can be run simultaneously (in parallel). It therefore behoves any performance driven computing task to take advantage of this to its fullest extent. This is termed High Performance Computing (HPC), some techniques of which will be discussed in the remainder of this chapter.

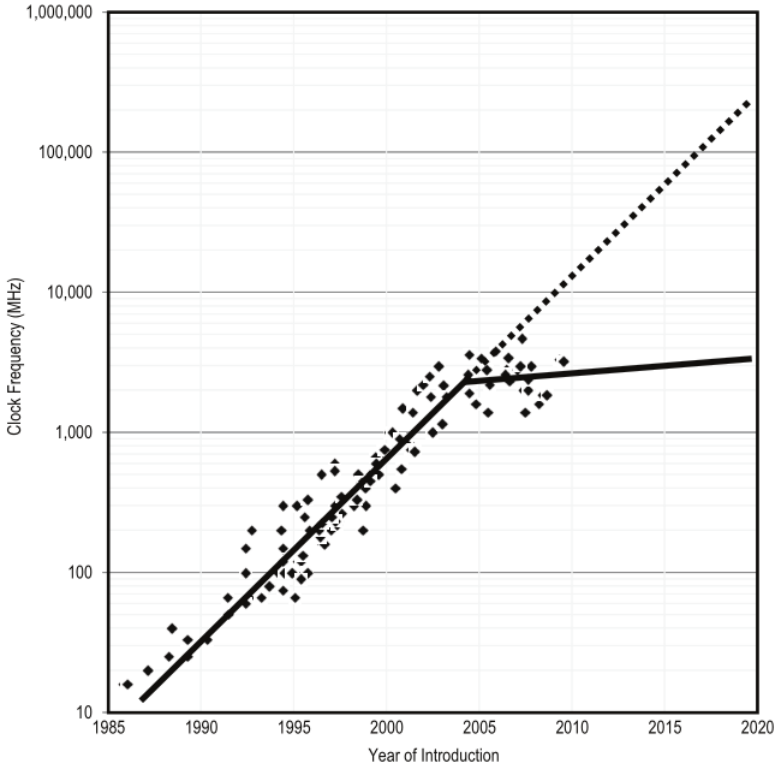


Figure 2.1: Historical performance of single processor performance [2].

2.2 High Performance Computing (HPC) techniques

Despite common misconception, high performance computing is not relegated to computer clusters (super computers). A computer cluster refers to many computers networked together to combine their performance. While it may have been the case that HPC was relegated to such clusters, with the advent of multiple processors in all but the weakest computers, the multiple processors are in fact a mini cluster. Thus, the same techniques that might have only been applied to multiple connected computers can be carried out on a single one. Naturally, the scale of problems will be smaller but it is still an important fact nonetheless, as it impacts the modularity of the software design approach.

Central to the philosophy of high performance computing is the ability to complete tasks in parallel. Therefore, all the relevant techniques will focus on efficiently creating and managing an environment where this can be accomplished. The creation and management of such parallel environments carries significant computational cost, the cost depending on both scale and implementation. It is therefore important to carefully analyse the problem and select the most suitable tool for the job.

Two software tools will be discussed, namely Message Passing Interface

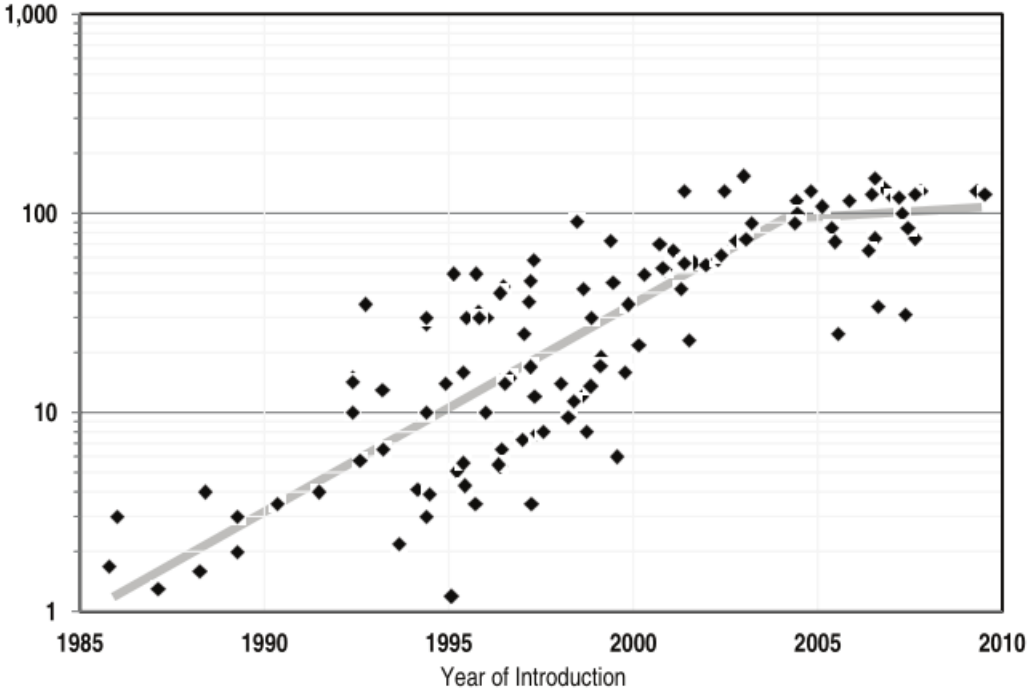


Figure 2.2: Microprocessor heat dissipation (watts) from 1985 to 2010 [2].

(MPI) and Open Multi Processing (OpenMP). While these tools are indeed powerful, it would be remiss not to also include additional, more hardware based means, such as graphical processing units (GPU's).

2.2.1 Useful Terminology

- **Core** - A processing unit within a processor.
- **Single Core Processor** - A processor containing a single processing unit (core).
- **Multi Core Processor** - A processor which contains multiple processing units (multiple cores).
- **Node** - A single computer, powered by multi core processors.
- **MPI Process** - A parallel instance, commonly matched to a single core.
- **Thread** - A sequence of computer instructions.
- **Thread Block** -A group of threads.

2.2.2 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standard set to govern the core programming library routines (API) for distributed system parallel programming. There are both free (MPICH [13]) and commercial (Intel MPI [15]) implementations of this API, which while extensive at around 186 different function calls, the premise is quite simple. An arbitrary amount of MPI processes are spawned, all which operate in parallel. Stated in [16], MPI is based on the Single Program Multiple Data (SPMD) paradigm. This implies that each MPI process runs the same code without access to shared memory. While each process is independent, they are capable of interprocess communication via the MPI API. The ability to network many nodes together allows for scalability and is illustrated in Figure 2.3.

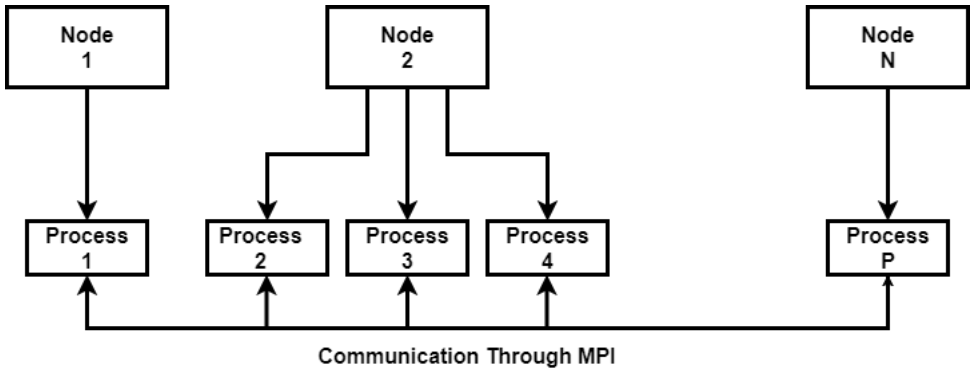


Figure 2.3: A simple example of a distributed memory system using MPI for communication.

While simple, there is a certain finesse in implementing an MPI algorithm in order to utilise what the MPI API offers effectively. As with any high performance technique, there are caveats and when ignored can cause the opposite of the desired effect.

2.2.2.1 Communication cost and the amount of MPI processes

As mentioned previously, while an arbitrary amount of processes can be spawned, best results are generally garnered when the number of available cores exceeds the number of processes. The work required to be performed for solving the problem will generally be split as equally as possible among the processes. An easy trap to fall into is spawning an over abundance of processes. This brings up the most critical point of note when using MPI i.e. the cost of interprocess communication. While perfectly parallel programs, which require no communication between processes do exist, a majority of problems require at least some amount of communication to function effectively. A cost benefit analysis must be done to ensure that the costs of communication do not overshadow

the speed gain from the processes acting in parallel. It is therefore critical to evaluate the algorithmic structure of the problem and apply resources effectively.

2.2.3 Open Multi Processing (OpenMP)

Open Multi Processing (OpenMP) is a high level parallelism specification for library routines and compiler directives [14]. OpenMP is based on a Shared Memory Programming (SMP) approach which unlike MPI has access to shared memory. This approach allows for parallel regions to be spawned at will when necessary in a program. As seen in Figure 2.4, the program is only parallel in certain regions. This parallel region spawns multiple threads which will have access to the same memory and operate independently. While threads have no communication cost due to shared memory access, there are costs involved for creating the parallel region, as well as for the management of the computer processor for the threads. It is then up to the implementer to weigh the costs of creating a parallel region versus the amount of work being done within it.

The amount of threads specified for the parallel region is governed by the processor on which the work is executed. Generally, a processor can only run a finite amount of threads in parallel and as the number of threads are increased beyond that point, a considerable management overhead diminishes the performance of the program. It is therefore necessary to be familiar with the specifications of the processor and adjust the number of threads accordingly.

2.2.4 A hybrid parallelism approach using MPI and OpenMP

As discussed in the previous section, MPI focusses on interprocess parallelism, while OpenMP is focused on intraprocess parallelism. From this, it follows that an effective parallelism strategy is to exploit the scaling of MPI with multiple

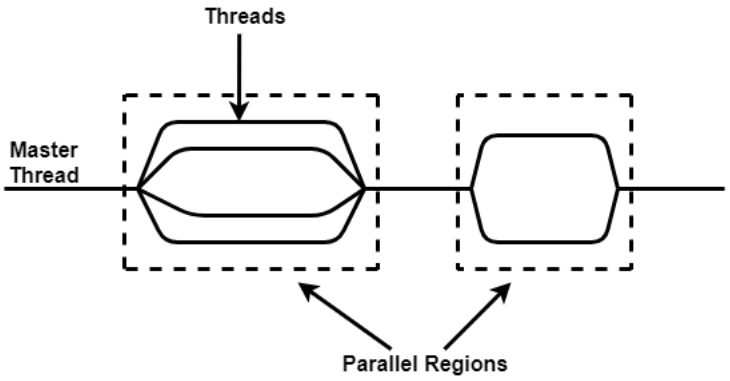


Figure 2.4: A simple overview on parallelism using OpenMP

processes over multiple nodes while applying more focused local parallelism using OpenMP.

Consider a problem parallelised using MPI. As the number of processes increase, the speed-up begins to peter off due to communication cost compared to the work done on the problem. It will be prudent to then apply OpenMP parallelism as the speed-up of MPI begins to deteriorate.

However, the important consideration of OpenMP's thread management creation and cost should be taken into account. A single core within a processor is only able to support one thread effectively. An overallocation of threads will produce diminished results, so it is important to understand the platform used. A simple formula to determine whether overallocation occurs is as follows,

$$\text{number of OpenMP threads} > \frac{\text{total cores available}}{\text{number of MPI processes}}. \tag{2.1}$$

An example using this formula: if there are 24 cores available and 8 MPI processes, more than 3 OpenMP threads would reduce the amount of speed-up. In contrast to a core, which ideally supports a single thread, a graphical processing units supports thousands, and will be discussed next.

2.2.5 Graphical Processing Units (GPU's)

Mentioned previously is the applicability of additional hardware based parallelism where Graphical Processing Units (GPU's) play a central role. Containing hundreds of processing units, GPU's have gained significant traction in the HPC field. The main premise behind the parallelism incorporated by GPU's is Single Instruction Multiple Threads (SIMT), meaning that a single set of instructions called a kernel are defined, which threads execute in parallel. These kernels are written in a specific GPU compatible language such as OpenCL [17] or CUDA [18], with the principal difference being that OpenCL is non-propriety while CUDA is only usable by NVIDIA GPUs. Kernels can also be generated dynamically, using a framework such as OpenACC [19], which introduces parallelism with compiler directives. However, dynamic kernels are not as effective for complex tasks.

Illustrated in Figure 2.5 is a graphical illustration of the parallelism applied in a GPU. A grid is created which is filled with thread blocks. Thread blocks as well as the threads within all operate in parallel. The onus is on the calling routines (i.e. the main/host software that calls the GPU parallelisation) to select the appropriate grid and thread block sizes to ensure optimal performance. While this choice is mainly based on the structure of the problem, the hardware of the GPU used also needs consideration with regards to its limitations. Lower end GPU's tend to have a cap on the size of the thread blocks and it is therefore critical to dynamically allocate these if the program is required to run on a variety of platforms.

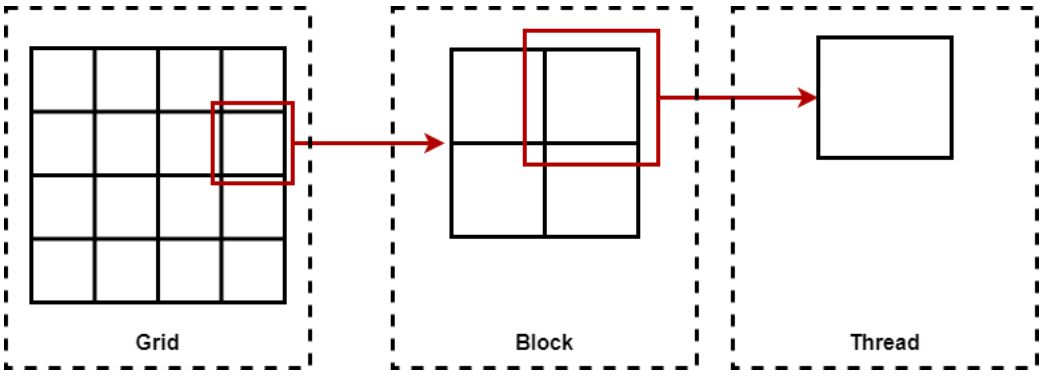


Figure 2.5: A diagram illustrating the grid, blocks and threads of a GPU.

A very significant overhead is observed when transferring data to and from a GPU. As with MPI and OpenMP, a cost benefit analysis is required to evaluate whether it is feasible to parallelise a problem with a GPU. This is clearly illustrated in Figure 2.6(with results based on that obtained from [3]), that as the amount of data transferred increases, so too does its dominance on execution time. Highlighted in [3] is that for the GTX 480, the data transfer time makes up 72% of the total execution time when considering 64 M 32 bit keys. A less pronounced effect is seen in the Tesla C2050 (a faster GPU than the GTX 480), that further highlights the importance of being aware of the GPU specifications as it can have a significant effect on performance.

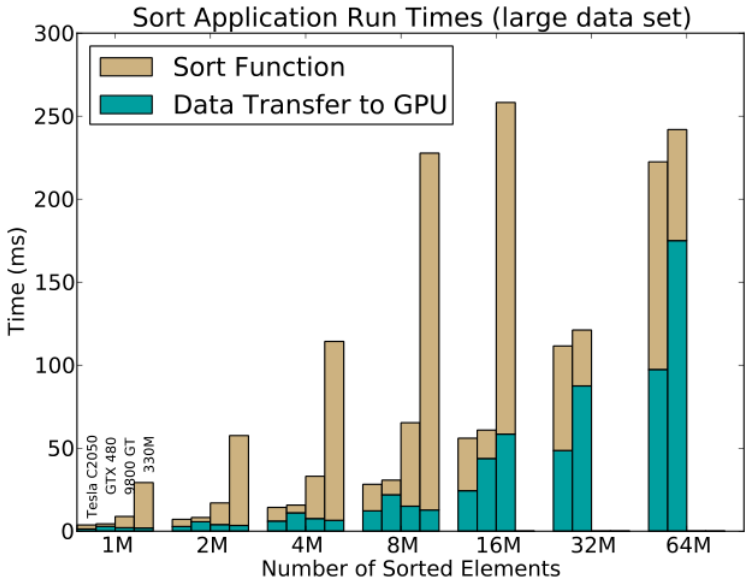


Figure 2.6: A graph comparing the data transfer and execution times of different GPU's and problem sizes [3]

2.3 The Lengau cluster at the Centre for High Performance Computing (CHPC)

The Centre for High Performance Computing (CHPC) hosts multiple computer clusters which are available for African researchers with the aim of uplifting computationally intensive research in the fields of science and engineering [20].

The author was granted access to the CHPC's Lengau cluster for the experiments shown in this work. The cluster consists of 1368 nodes each with two Intel Xeon CPU E5-2690 @ 2.60GHz CPUs (12 cores per CPU and therefore 24 cores per node) networked using the Infiniband interconnect. Also available in the cluster is the use of a Tesla V100 GPU. The detailed information for the node and GPU are found in Appendices A and B respectively.

2.4 Applying HPC techniques to domain decomposition methods

Previously, in Chapter 1, concern was raised regarding the issues of speed and memory for electrically large CEM problems. HPC lends itself naturally to the improvement of these computational costs. Memory is shared among many parallel instances and due to parallelism, the speed is naturally increased. Consider Equation (1.10), the blocked form of the MoM matrix equation. These matrix blocks of $[Z]$ and $[V]$ can be evaluated independently in parallel. Solving for $[I]$, however, depends on the data from multiple blocks simultaneously. The nature of the dependence varies between domain decomposition methods and as such, an HPC solution requires individual tailoring. The specifics of the application of the HPC techniques mentioned in this chapter will be further elaborated on later in this work.

2.5 Conclusion

In this chapter, two software based high performance techniques (MPI and OpenMP) were discussed. A hybridisation of the two was also explained and the common pitfalls were explored with regards to the costs of implementing said methods. A hardware based parallelisation method using GPU's, was also detailed. In the next chapter, the author's CEM solver, CEMACS, will be discussed.

Chapter 3

CEMACS

CEMACS is a C++ CEM suite developed by the author for research and development of highly performant domain decomposition methods based on the MoM. This chapter will first present an overview of CEMACS followed by considerations made for the implementation. Following this, methods of evaluation will be discussed regarding the speed and accuracy of the software.

3.1 An overview of CEMACS

CEMACS consists of three parts, namely premacs, the pre-processor, cemacs, the kernel, and postmacs, the post-processor. Illustrated in Figure 3.1 is how the three components are connected. Geometry is read by the pre-processor and a *.kif* file containing geometry, attributes and basis function data is written. Thereafter, the kernel operates on the input data using the methods explained in the following chapters. A *.kof* file containing the solution data is then written and used as input to the post-processor where it is processed. These steps will be explained in further detail below.

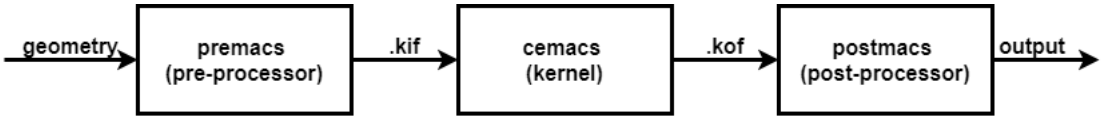


Figure 3.1: A high level flow diagram of CEMACS.

3.1.1 On the use of Altair FEKO

Altair FEKO [21] is a highly regarded commercial CEM software suite which provides a comprehensive CAD modeller, CEM solvers and the calculation and display of post-solution metrics. As CEMACS is a software used for research and development, it is critical to verify it against a reference solution. FEKO, in the context of CEMACS is used in the following ways:

- Create and mesh geometry.
- Calculate post-solution metrics such as fields.

3.1.2 Premacs

Premacs is a python pre-processor which reads in the geometry and assigns the necessary basis functions. The geometry, basis functions and necessary data, such as the frequency and excitation parameters are written to a *.kif* file which is then read by the kernel. The use of a pre-processor allows the kernel to operate on one file format rather than the various formats found in various CAD modellers. Currently, only the FEKO *.out* file is a supported geometry input but it is straightforward to add support for other formats.

3.1.3 Cemacs, the CEMACS kernel

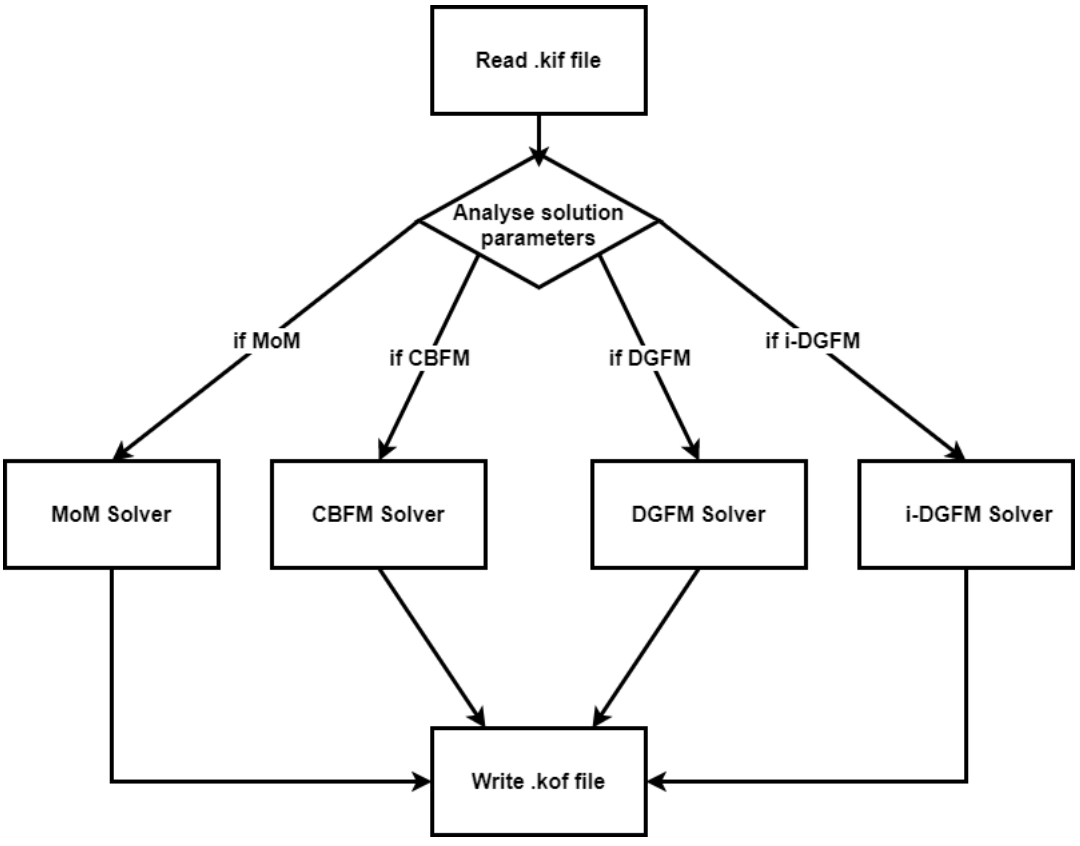


Figure 3.2: An overview of cemacs, the CEMACS kernel.

As shown in Figure 3.2, cemacs, the C++ kernel, consists of four solvers, namely, the MoM, CBFM, DGFM and the i-DGFM. After the *.kif* file is read, the parameters are analysed and a solver is chosen. After the solution is

complete, a *.kof* file is written containing the solution data. This procedure is identical for the serial, MPI and CUDA build targets. The solvers, as well as their methods of parallelisation, will be discussed in the following chapters.

3.1.3.1 Linear Algebra Libraries

Integral to any of the solvers implemented is the need of linear algebra routines to complete matrix and vector operations. One such routine required is the LU-decomposition as mentioned in Chapter 1. The routines needed can be manually written, but it is more efficient to take advantage of the numerous linear algebra libraries available. Table 3.1 lists the libraries and how they are used.

Library	Usage
LAPACK [22]	Serial LU-decomposition and matrix-matrix, and matrix-vector multiplication.
OpenBLAS [23]	OpenMP threaded LU-decomposition and matrix-matrix, and matrix-vector multiplication.
cuBLAS [24]	matrix-matrix and matrix-vector multiplication for CUDA
cuSOLVER [25]	LU-decomposition for CUDA

Table 3.1: Linear algebra libraries used in CEMACS

3.1.4 Postmacs

Finally, postmacs, the python post-processor reads in the *.kof* file and compares the solution to the one obtained by FEKO. There is also the option to write the solution to a FEKO *.str* file which can be read and used by FEKO to calculate post solution metrics. Since postmacs has access to both the geometry and the solution, it is possible to calculate and display post-solution metrics if needed.

3.2 Considerations regarding high performance computing in CEMACS

3.2.1 MPI load balancing

One of the critical aspects of correct MPI usage is the balancing of the workload between all the processes. Improper load balancing can lead to processes waiting on one another for data resulting in deteriorating efficiencies. Consider the simple scenario of operating on four datasets with two processes shown in Figure 3.3. The load imbalance is clear: Process 1 is left idle for two thirds of the total runtime. It is easily noticeable that if Process 1 operated on Dataset 3 as well, total runtime would decrease by a third.

Fortunately, the block like nature of the matrix equation (Equation (1.10)) used in domain decomposition methods makes it rather trivial to balance. The

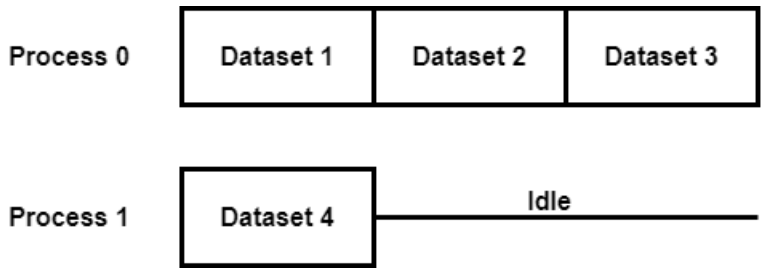


Figure 3.3: A simple diagram showing load imbalance between two processes.

rows in the matrix equation need to be divided as equally as possible and a simple formula for this, using integer division, is

$$\text{Number of rows on process} = \frac{\text{Total number of rows} + \text{Process rank}}{\text{Total number of processes}} \tag{3.1}$$

where the process rank is the integer identifier of the current process (numbering from 0 to M, with M being the number of processes).

3.2.2 Parallelism in CUDA

A brief explanation regarding the parallelism present in CUDA was presented previously in Section 2.2.5. In the context of this thesis an optimal grid would be one where number of threads = number of elements operated on. Due to the constraints mentioned below, there are cases where the number of threads created by the grid is more than required. A near optimal grid can therefore be created my minimising the amount of excess threads and is created using

$$\begin{aligned} \text{grid.x} &= \frac{\text{number of matrix rows} + \text{block.x} - 1}{\text{block.x}} \\ \text{grid.y} &= \frac{\text{number of matrix cols} + \text{block.y} - 1}{\text{block.y}} \end{aligned} \tag{3.2}$$

where grid and block are vectors defining the size of the grid and blocks respectively. Therefore, .x refers to the number of rows and .y the number of columns in either the size of the grid or blocks. The block size i.e. block.x $\times$ block.y must always be a multiple of 32 due to the nature of the SIMT architecture which manages and executes threads in groups of 32 called warps. Therefore, unless the number of row/columns of a matrix is a multiple of 32, there will always be a few extra threads. It is therefore necessary to check for these extra threads to avoid writing data outside the matrices' allocated memory. This concept is illustrated in Figure 3.4 when considering the arbitrary matrix (in black) with thread blocks (in red) overlapping as shown. The excess threads (dotted red cross) need to be filtered to avoid memory errors.

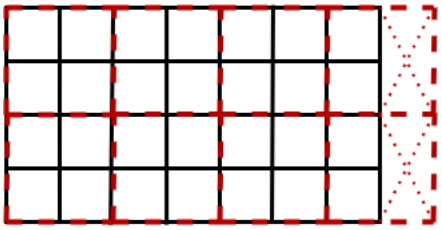


Figure 3.4: An arbitrary matrix operated on by an arbitrary block size showing the excess threads.

3.3 Figures of merit - Evaluating CEMACS

Mentioned previously was the need to evaluate the results of CEMACS by comparison to reference solutions. Regarding accuracy, CEMACS will be measured against FEKO using a normalised error percentage metric defined below. With respect to parallelisation, CEMACS will be compared to the *ideal* speed-up, i.e. the best speed-up possible, explained below in Section 3.3.2. Lastly, a method will be described in order to critically evaluate the speed-up of GPU versus CPU implementations.

3.3.1 Relative Error

A normalised error percentage, $\epsilon_{\%}$, from [12], can be calculated using the Euclidean norm of vectors as

$$\epsilon_{\%} = \sqrt{\frac{\sum_{n=1}^{N} |x_n^{REF} - x_n|^2}{\sum_{n=1}^{N} |x_n^{REF}|^2}} \times 100\% = \frac{||x_n^{REF} - x_n||_2}{||x_n^{REF}||_2} \times 100\% \tag{3.3}$$

where x is the vector tested and x_n^{REF} the reference vector.

3.3.1.1 Relative error when applied to sinusoidal results

The relative error calculated above cannot be applied to results that exhibit a sinusoidal nature as the error is calculated relative to zero. To rectify this, the error will be calculated using

$$\epsilon_{\%} = \frac{max(x_n^{REF} - x_n)}{x_{peak\_to\_peak}^{REF}} \times 100\% \tag{3.4}$$

where x is the vector tested and x_n^{REF} the reference vector.

3.3.2 Speed-up

From [7] speed-up for a fixed problem size, S is calculated as

$$S = \frac{T_s}{T_p} \tag{3.5}$$

with T_s and T_p are the times taken to run the program in serial and in parallel respectively. While the *ideal* speed-up is naturally $S = P$, P being the number of parallel instances, a more realistic calculation, given by Amdahl's law, would be

$$S = \frac{1}{1 - \zeta + \frac{\zeta}{P}}, \tag{3.6}$$

where ζ is the percentage amount of the program which benefits from parallelism. In the context of CEMACS, the MPI implementation parallelises $0.97 < \zeta < 0.99$ (obtained using runtime measurements) of the program depending on the specific solution method used. Therefore, for the sake of consistency between evaluations, speed-up will be compared to $S = P$. It is also important to note that both the speed-up proposed by Amdahl as well as the ideal speed-up neglect the cost of the parallel implementation such as communication or thread creation and management. It is then expected that the results obtained will be less than ideal as these costs become more apparent as the number of parallel instances rise.

3.3.2.1 Special consideration regarding GPU speed-up

Since the GPU operates using a SIMT architecture, speed-up measured for a fixed problem size does not provide much insight. Consider the following simple code snippets, in a CUDA kernel where an arbitrary calculation is made on each element of a vector. The index of a thread can be obtained using variables provided by the CUDA library as shown in Figure 3.5.

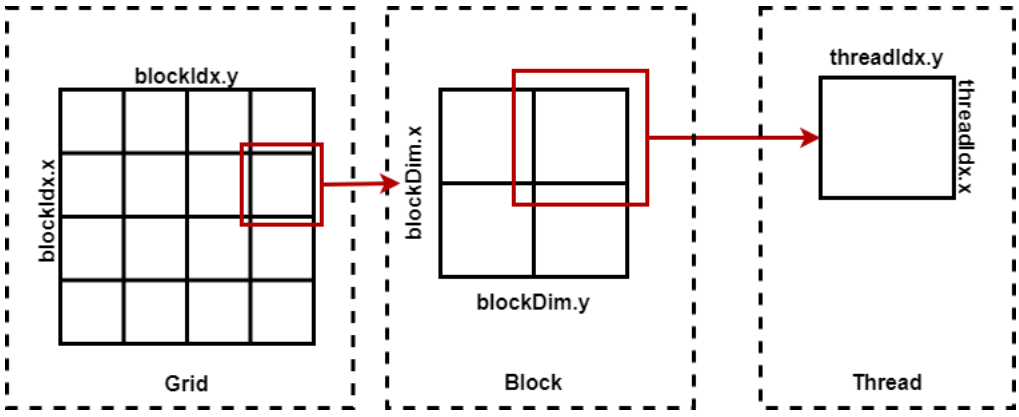


Figure 3.5: CUDA indexing of grids, blocks and threads.

Using this, the amount of threads that operate in parallel can be controlled, for example

```
int thread_index = blockIdx.x * blockDim.x + threadIdx.x;
int thread_span  = (problem_size + thread_index ) /
                    total_number_of_threads;
```

```
for (int i = thread_index; i < thread_span; i++)
{
    vector[i] = some_arbitrary_calculation();
}
```

where total_number_of_threads < problem_size. This is however not optimal and typically, for a problem of this nature, there would be one thread per vector element as

```
int thread_index = blockIdx.x * blockDim.x + threadIdx.x;

if (thread_index < problem_size) // Check for excess threads
{
    vector[thread_index] = some_arbitrary_calculation();
}
```

3.4 Conclusion

This chapter discussed CEMACS, the author's CEM solver. A high level overview was displayed followed by specific considerations made with regard to high performance computing. Finally, methods of evaluating the speed and accuracy of the software were formulated. In the following chapters the domain decomposition solvers mentioned will be implemented, validated and parallelised. However, in the next chapter, a formulation of the Method of Moments (MoM), the foundation of the domain decomposition methods will be presented.

Chapter 4

The Method of Moments

This chapter will discuss the Method of Moments (MoM) starting with its formulation followed by a discussion on its implementation. Finally, the MoM, as implemented in CEMACS, will by applied to two problems and evaluated in terms of accuracy.

4.1 MoM Formulation

As mentioned in Chapter 1, when using the MoM, the surface of the geometry is replaced by equivalent surface currents which are then discretised. In this work, discretisation will occur by means of triangular patches, named Rao-Wilton-Glisson (RWG) triangular patches as presented in [4]. Triangular patches were chosen as they have the ability to conform with any surface, and along with the Electric Field Integral Equation (EFIE), are used to formulate an efficient MoM algorithm, which is a summary based on the work done in [4].

4.1.1 Electric Field Integral Equation Formulation

If a time harmonic electric field $\mathbf{E}^{inc}$ induces surface currents, $\mathbf{J}$, on an open surface S, the normal component of $\mathbf{J}$ vanishes at its boundaries. The scattered electric field $\mathbf{E}^{scat}$ is then calculated as

$$\mathbf{E^{scat}} = -j\omega\mathbf{A} - \nabla\Phi \tag{4.1}$$

with the magnetic vector potential $\mathbf{A}$ defined as

$$\mathbf{A}(\mathbf{r}) = \frac{\mu}{4\pi}\int_S \mathbf{J}\frac{e^{-jkR}}{R}dS' \tag{4.2}$$

and the scalar potential defined as

$$\Phi(r) = \frac{1}{4\pi\epsilon}\int_S \sigma\frac{e^{-jkR}}{R}dS'. \tag{4.3}$$

The wave number k in both (4.2) and (4.3) is calculated as $k = 2\pi/\lambda$. λ is the wavelength and μ and ϵ are the permeability and permittivity of the material. Furthermore, R is the distance between the observation point $\mathbf{r}$ and the source point $\mathbf{r}'$ is calculated as,

$$R = |\mathbf{r} - \mathbf{r}'|. \tag{4.4}$$

The vector $\mathbf{r}$ is arbitrarily chosen while $\mathbf{r}'$ is a point on S, the surface. The following equation of continuity relates σ, the surface charge density to $\nabla_s \cdot \mathbf{J}$, the surface divergence of $\mathbf{J}$,

$$\nabla_S \cdot \mathbf{J} = -j\omega\sigma. \tag{4.5}$$

The boundary condition $\hat{\mathbf{n}} \times (\mathbf{E^{inc}} + \mathbf{E^{scat}}) = 0$ is then used to define an integrodifferential equation for $\mathbf{J}$. With $\hat{\mathbf{n}}$ the unit normal on S,

$$-\mathbf{E^i_{tan}} = (-j\omega\mathbf{A} - \nabla\Phi)_{tan}, \text{ with } r \text{ on } S. \tag{4.6}$$

Equation (4.6) is thus the EFIE and relates the unknown surface currents $\mathbf{J}$ to the incident field $\mathbf{E}^{inc}$. Solving Equation (4.6) requires basis functions to represent the unknown surface currents which will be discussed next.

4.1.2 Basis Function Development

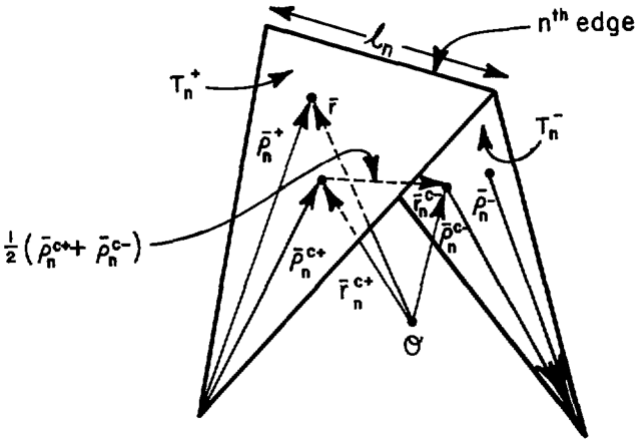


Figure 4.1: A pair of triangles with parameters required for the MoM [4].

The basis functions developed in this section are required to be both suitable for EFIE usage as well as the RWG triangular patches. The assumption is made that each basis function in the patch model is associated with a non-boundary interior edge and evaluates to zero except in the two triangular patches which share the edge. Shown in Figure 4.1, the *nth* edge is shared by triangles T_n^+ and T_n^-, the points in either chosen as the position vectors $\vec{r}$,

defined in relation to the global coordinate origin or $\vec{\rho}_n^{\pm}$, defined with respect to the free vertex of $T_n^{\pm}$. Triangles in the pair will be designated as either positive or negative for the *nth* edge based on the assumption that current flows from the positive to the negative triangle. From this, $\vec{f}_n(\vec{r})$, the vector basis function associated with the *nth* edge is defined as

$$\vec{f}_n(\vec{r}) = \begin{cases} \frac{l_n}{2A_n{}^+}\vec{\rho_n}^+ & \vec{r} \text{ in } T_n{}^+ \\ \frac{l_n}{2A_n{}^-}\vec{\rho_n}^- & \vec{r} \text{ in } T_n{}^- \\ 0 & otherwise, \end{cases} \tag{4.7}$$

with its surface divergence being

$$\nabla_s \cdot \vec{f}_n(\vec{r}) = \begin{cases} \frac{l_n}{A_n{}^+} & \vec{r} \text{ in } T_n{}^+ \\ -\frac{l_n}{2A_n{}^-} & \vec{r} \text{ in } T_n{}^- \\ 0 & otherwise. \end{cases} \tag{4.8}$$

An approximation of the current in terms of $\vec{f}_n$ on the surface S is then given by

$$\mathbf{J} \cong \sum_{n=1}^{N} I_n \vec{f}_n(\vec{r}) \tag{4.9}$$

with N the number of non-boundary interior edges. From [4], I_n in Equation (4.9) is the normal component of the current density passing over the *nth* edge. This is due to $\vec{f}_n$ at the *nth* edge being unity.

4.1.3 Applying the Testing Procedure

The method of transforming the EFIE into a linear equation set is done by choosing the basis functions $\mathbf{f_n}$ as the testing functions as shown in [4]. Equation (4.6) is tested with $\mathbf{f_m}$ using

$$\langle \mathbf{f}, \mathbf{g} \rangle \equiv \int_S \mathbf{f} \cdot \mathbf{g} dS, \tag{4.10}$$

resulting in

$$\langle \mathbf{E^{inc}}, \mathbf{f_m} \rangle = j\omega \langle \mathbf{A}, \mathbf{f_m} \rangle + \langle \nabla\Phi, \mathbf{f_m} \rangle. \tag{4.11}$$

The last term in Equation (4.11) is then be defined as

$$\langle \nabla\Phi, \mathbf{f_m} \rangle = -\int_S \Phi \nabla_S \cdot \mathbf{f_m} dS \tag{4.12}$$

using a vector calculus identity. Further, an approximation using Equation (4.8) is given as

$$\begin{aligned} \int_S \Phi \nabla_S \cdot \mathbf{f_m} dS &= l_m \left(\frac{1}{A_m{}^+} \int_{T_m{}^+} \Phi dS - \frac{1}{A_m{}^-} \int_{T_m{}^-} \Phi dS\right) \\ &\cong l_m [\Phi(r_m{}^{c+}) - \Phi(r_m{}^{c-})]. \end{aligned} \tag{4.13}$$

Using a similar approximation, that $\mathbf{E^{inc}}$ in a triangle is its value at the centre of the triangle, Equation (4.11) is be shown to be

$$\left\langle \begin{Bmatrix} \mathbf{E^i} \\ \mathbf{A} \end{Bmatrix}, \mathbf{f_m} \right\rangle = l_m \left[\frac{1}{2{A_m}^+} \int_{{T_m}^+} \begin{Bmatrix} \mathbf{E^i} \\ \mathbf{A} \end{Bmatrix} \cdot {\rho_m}^+ dS + \frac{1}{2{A_m}^-} \int_{{T_m}^-} \begin{Bmatrix} \mathbf{E^i} \\ \mathbf{A} \end{Bmatrix} \cdot {\rho_m}^- dS \right]$$
$$\cong \frac{l_m}{2} \left[\begin{Bmatrix} \mathbf{E^i}(\mathbf{{r_m}^{c+}}) \\ \mathbf{A}(\mathbf{{r_m}^{c+}}) \end{Bmatrix} \cdot {\rho_m}^{c+} + \begin{Bmatrix} \mathbf{E^i}(\mathbf{{r_m}^{c-}}) \\ \mathbf{A}(\mathbf{{r_m}^{c-}}) \end{Bmatrix} \cdot {\rho_m}^{c-} \right]. \tag{4.14}$$

Finally, Equation (4.11) can be rewritten based on the approximations made

$$j\omega l_m \left[\mathbf{A}(\mathbf{{r_m}^{c+}}) \cdot \frac{{\rho_m}^{c+}}{2} + \mathbf{A}(\mathbf{{r_m}^{c-}}) \cdot \frac{{\rho_m}^{c-}}{2} \right] + l_m [\Phi({r_m}^{c-}) - \Phi({r_m}^{c+})]$$
$$= l_m \left[\mathbf{E^i}(\mathbf{{r_m}^{c+}}) \cdot \frac{{\rho_m}^{c+}}{2} + \mathbf{E^i}(\mathbf{{r_m}^{c-}}) \cdot \frac{{\rho_m}^{c-}}{2} \right]. \tag{4.15}$$

The approximations made are only valid due to the potentials being locally smooth within their domains and it is therefore important to note that the impedance matrix $[Z]$, discussed later, is unsymmetrical due to this.

4.1.4 Derivation of the MoM matrix equation

An $N \times N$ system of linear equations are obtained by substituting (4.9) into (4.15). Written in matrix form,

$$[Z][I] = [V], \tag{4.16}$$

with the dimensions of $[Z]$ being $N \times N$ and both $[I]$ and $[V]$ being $N \times 1$. The individual elements of matrix $[Z]$ is given by

$$Z_{mn} = l_m \left[j\omega \left(\mathbf{{A_{mn}}^+} \cdot \frac{{\rho_m}^{c+}}{2} + \mathbf{{A_{mn}}^-} \cdot \frac{{\rho_m}^{c-}}{2} \right) + {\Phi_{mn}}^- - {\Phi_{mn}}^+ \right] \tag{4.17}$$

and the elements of $[V]$ as

$$V_m = l_m \left(\mathbf{{E_m}^+} \cdot \frac{{\rho_m}^{c+}}{2} + \mathbf{{E_m}^-} \cdot \frac{{\rho_m}^{c-}}{2} \right), \tag{4.18}$$

with

$$\mathbf{{E_m}^\pm} = \mathbf{E^i}(\mathbf{{r_m}^{c\pm}}) \tag{4.19}$$

The magnetic vector potential and the scalar potential are then defined as

$$\mathbf{{A_{mn}}^\pm} = \frac{\mu}{4\pi} \int_S \mathbf{f_n}(\mathbf{r'}) \frac{e^{-jk{R_m}^\pm}}{{R_m}^\pm} dS', \tag{4.20}$$

and

$$\Phi_{mn}{}^\pm = -\frac{1}{4\pi j\omega\epsilon} \int_S \nabla'_s \cdot \mathbf{f_n}(\mathbf{r'}) \frac{e^{-jk{R_m}^\pm}}{{R_m}^\pm} dS' \tag{4.21}$$

respectively, with

$${R_m}^\pm = |\mathbf{{r_m}^{c\pm}} - \mathbf{r'}|. \tag{4.22}$$

Given $[Z]$ and $[V]$, the unknown $[I]$ can then be solved using linear algebra.

4.1.5 Evaluating the MoM impedance matrix effectively

As mentioned in Chapter 1, the filling of the matrix, $[Z]$, takes a significant portion of time as N grows larger as it scales as $\mathcal{O}(N^2)$. Noted in [4] is that there are certain common integrals (Equations (4.20) and (4.21)) when calculating the entries of $[Z]$. It is then prudent to rather utilise a face-pair approach rather than the original edge-pair approach. The advantage of this can be quantified in the best case scenario, a geometry without boundary edges (e.g. a sphere) would only require the calculation of $4N^2/3$ integrals in the face-pair approach compared to $12N^2$ integrals in the edge pair approach. While there is a reduction of approximately 9 times, the integrals are calculated in groups of 3 garnering a more realistic speed increase of $3\times$.

Calculating these integrals, irrespective of the approach used, requires triangular domain numerical quadrature schemes. Described in [7] is the Gaussian quadrature scheme which is used in this work. Important considerations need to be made using this as the self terms in $[Z]$, Z_{mn} where $m = n$, need to be isolated and solved analytically using a singularity treatment method. Such methods are elaborated on in [26] and [27].

4.2 Implementing the MoM

The implementation of the MoM is illustrated in Figure 4.2. The calculation of $[Z]$ has been discussed above and the calculation of $[V]$, as well as and the solution of the matrix equation, i.e. (4.16) will be discussed next.

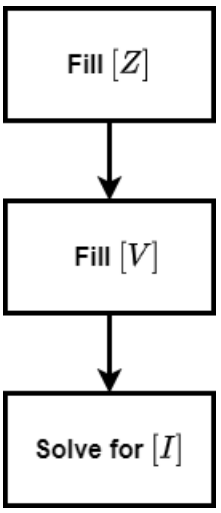


Figure 4.2: A block diagram showing the steps of the MoM algorithm.

4.2.1 Excitation Vector

Two classes of excitations, based on scattering and radiating problems will be discussed. In the case of a scattering problem, a plane wave is incident on the geometry while for a radiating problem, an excitation is applied through an edge feed. The calculation of $[V]$ will differ between them and this will be discussed below.

4.2.1.1 Scattering Problems

Using equation (4.19) the incident electric field is

$$\mathbf{E^i}(\mathbf{r}) = (E_\theta \hat{\theta_0} + E_\phi \hat{\phi_0}) e^{j\vec{k}\cdot\vec{r}} \tag{4.23}$$

for plane wave incidence as mentioned in [4] with the propagation vector $\vec{k}$ given as

$$\vec{k} = k(\sin\theta_0 \cos\phi_0 \hat{\mathbf{x}} + \sin\theta_0 \cos\phi_0 \hat{\mathbf{y}} + \cos\theta_0 \hat{\mathbf{z}}). \tag{4.24}$$

Using spherical coordinates, the vectors $\hat{\theta_0}$ and $\hat{\phi_0}$ describe the planes waves' angle of arrival and θ_0 and ϕ_0 are unit vectors coinciding with the usual spherical coordinate unit vectors in the direction of $\vec{k}$ from the global origin.

4.2.1.2 Radiating Problems

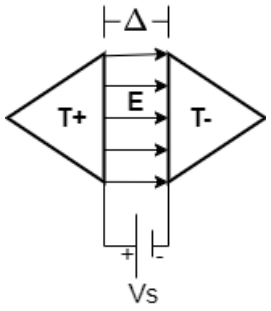


Figure 4.3: The delta feed model.

A feed model is introduced for radiating problems to account for the applied excitation. As discussed in [28], the delta-gap model is a technique whereby a gap of negligible width is applied as shown in Figure 4.3. As Δ tends to zero, an electric field $\mathbf{E}$ is created and is defined as

$$\mathbf{E} = V\hat{n}_y \tag{4.25}$$

using the delta function approximation. The elements of $[V]$ will be zero except at the common edges of the delta-gap model. Figure 4.3 shows the simplest

case and it is not uncommon to have more than two common edges. The value at each these common edges, m is given as

$$V_m = \int_{T^+ + T^-} \mathbf{E} \cdot \mathbf{f_m} dS = V \cdot \int_{T^+ + T^-} \hat{\mathbf{n}_y} \cdot \mathbf{f_m} dS. \tag{4.26}$$

As the normal component of $\mathbf{f_m}$ is always unity, Equation (4.26) can be simplified to

$$V_m = l_m V_s, \tag{4.27}$$

with l_m the length of the common edge.

4.2.2 Solving the MoM equation

Solving for $[I]$ in the MoM matrix equation is done using simple linear algebra

$$[Z]^{-1}[V] = [I]. \tag{4.28}$$

As mentioned previously, in all but the simplest problems, N gets quite large. The computational cost is excessive thus a more efficient technique, applying LU-decomposition, will be used as described below.

4.2.2.1 LU-decomposition

For the LU-decomposition method, the $[Z]$ matrix is factored into the product of a lower and upper triangular matrix

$$[Z] = [L][U]. \tag{4.29}$$

Substituted into the original MoM matrix equation, (4.16) now becomes

$$[L][U][I] = [V]. \tag{4.30}$$

Using a temporary variable, $[b] = [U][I]$, is substituted into (4.30), obtaining

$$[L][b] = [V]. \tag{4.31}$$

$[b]$ is then solved using forward substitution and consequently $[I]$ is solved using backward substitution. This process need not be done manually, a linear algebra library, in the case of this work, LAPACK [22], is used.

4.3 Numerical Results

This section will focus on applying the CEMACS MoM solver to a scattering and radiating antenna array in order to assess the accuracy of the solution. Mentioned previously in Subsection 3.1.1, FEKO will be used for comparison.

4.3.1 Applying CEMACS MoM to a square PEC plate array

Figure 4.4 illustrates an antenna array consisting of one hundred square plates simulated at 300 MHz. The E-field at the cut, shown as the red line is compared in Figure 4.5.

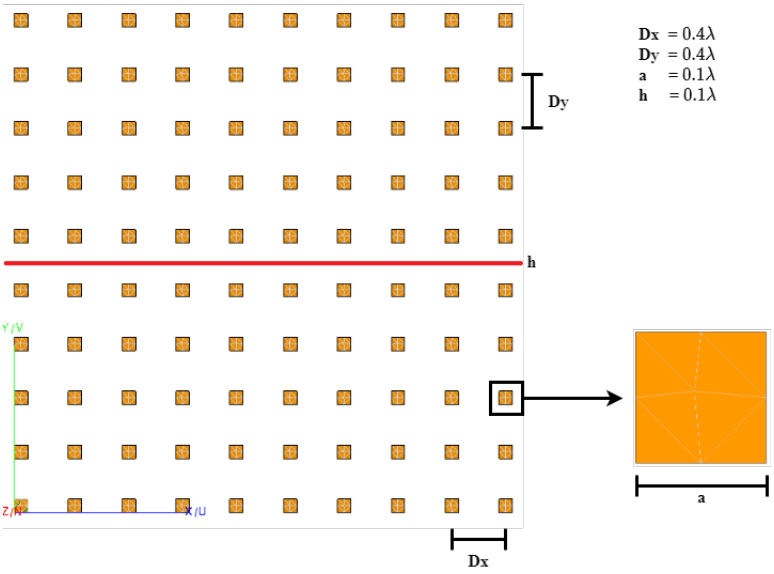


Figure 4.4: An antenna array consisting of one hundred square plates.

The error, as defined in Section 3.3.1.1, between CEMACS MoM and FEKO is $\epsilon = 6.39\%$ which is acceptable and can be reduced even further by taking into account the near singularities, i.e. when the source and observation points are near to each other. This result validates the accuracy of the CEMACS MoM solver.

4.3.2 Applying CEMACS MoM to a Vivaldi array

Next, consider the three element Vivaldi antenna array shown in Figure 4.6. Figure 4.7 shows the E-plane gain at 500 MHz and Figures 4.8 and 4.9 shows the magnitude and phase of S_{11} for the frequency range 500 MHz to 1 GHz.

Compared to FEKO, CEMACS MoM has a low ϵ as shown in Table 4.1. The error is again quite low. Differences can be attributed to the lack of near singularity treatment. CEMACS MoM is now validated for antennas with edge feeds such as mentioned in Subsection 4.2.1.2.

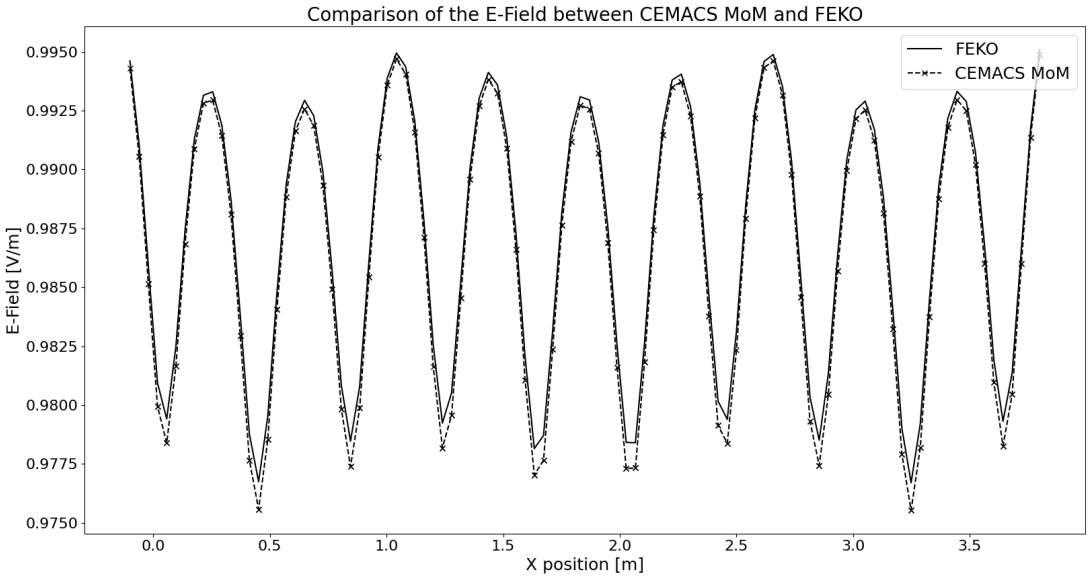


Figure 4.5: A comparison of the E-Field between CEMACS MoM and FEKO.

Figure 4.6: An antenna array consisting of three Vivaldi antennas.

Calculated Quantity	$\epsilon(\%)$
E-plane gain	1.60%
$\lvert S_{11}\rvert$	2.20%
$\angle S_{11}$	5.63%

Table 4.1: Relative error percentages ($\epsilon_{\%}$) between CEMACS MoM and FEKO for three calculated quantities.

4.4 Conclusion

In this chapter, the Method of Moments (MoM) was formulated and its implementation was discussed. Finally, the author's implementation, CEMACS MoM was validated within 6.39% against the commercial CEM suite FEKO. With the foundation of the domain decomposition methods presented, the next chapters will focus on three domain decomposition methods, starting with the Characteristic Basis Function Method (CBFM).

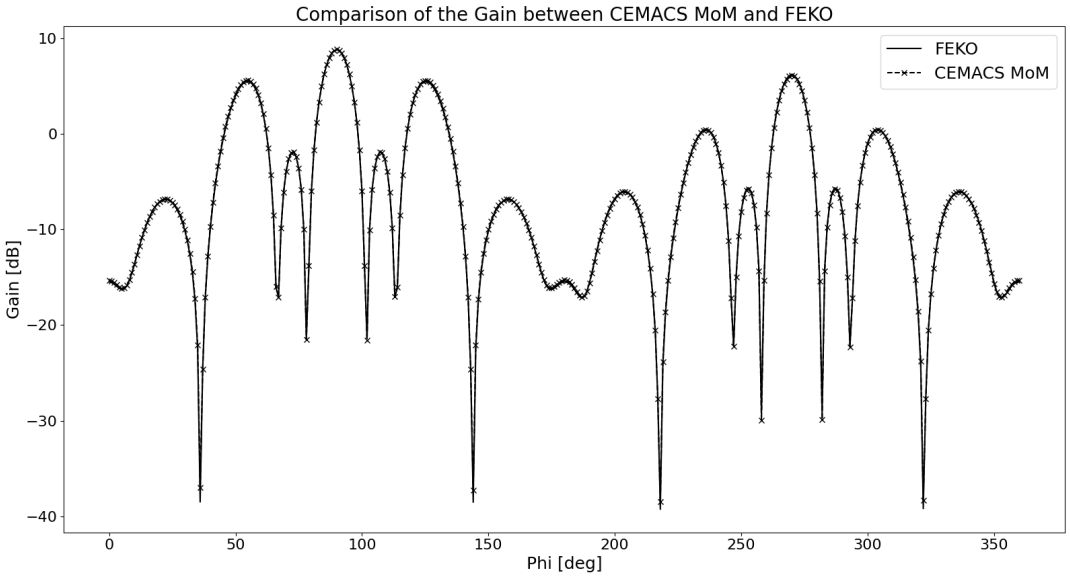


Figure 4.7: A comparison of the E-plane gain (dB) between CEMACS MoM and FEKO for a Vivaldi antenna array.

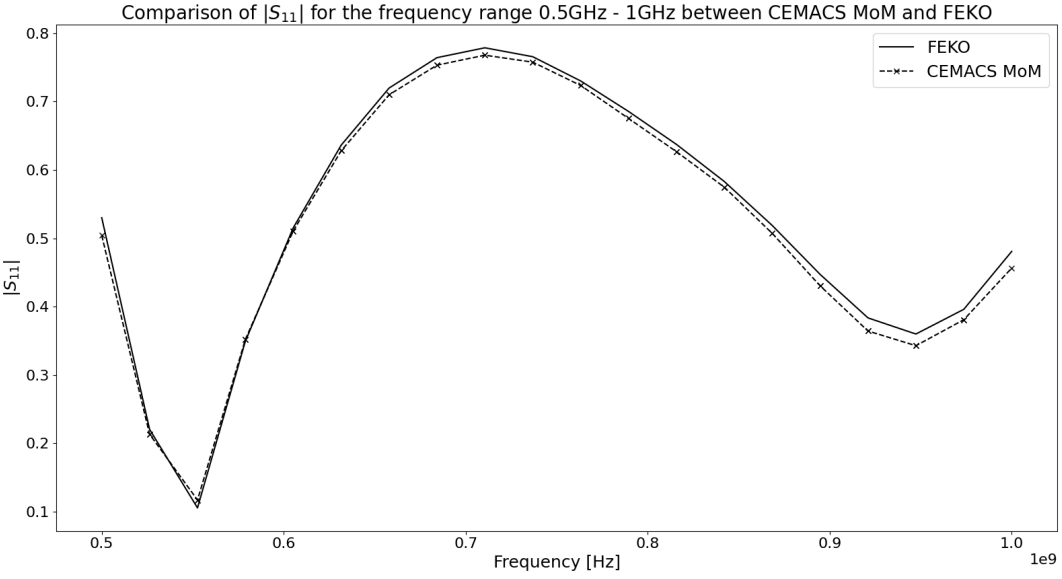


Figure 4.8: A comparison of $|S_{11}|$ between CEMACS MoM and FEKO for a Vivaldi antenna array.

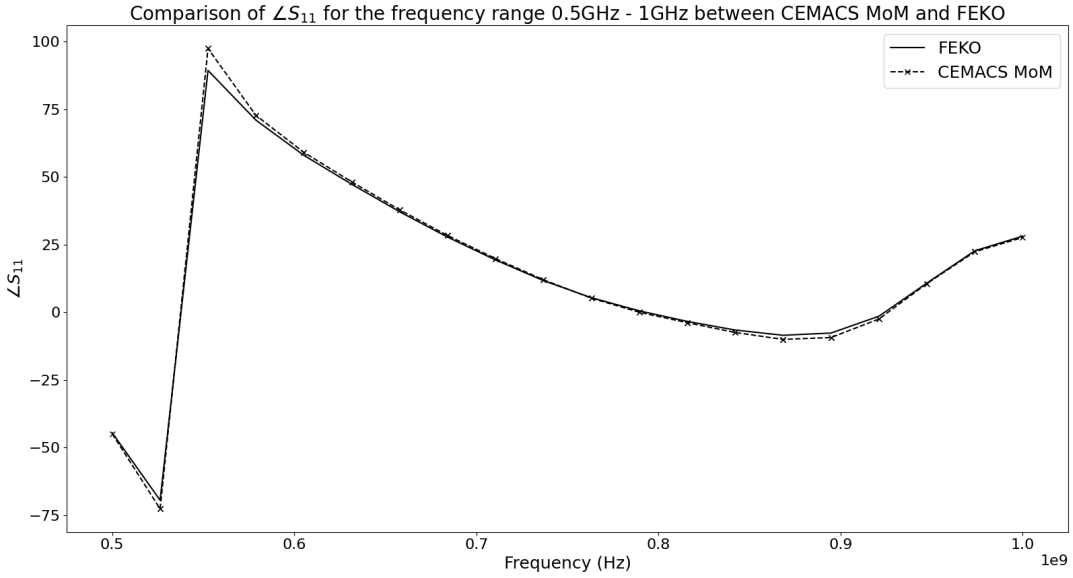


Figure 4.9: A comparison of the phase of S_{11} between CEMACS MoM and FEKO for a Vivaldi antenna array.

Chapter 5

The Characteristic Basis Function Method

As mentioned in Section 1.1.2.1 , as the number of RWG basis functions (N) increases, so does the computational cost of the MoM, as discussed in Chapter 4. Domain decomposition techniques can be used to alleviate this. One such technique is the Characteristic Basis Function Method (CBFM). Discussed in [10], the CBFM introduces characteristic basis functions (CBFs) in order to reduce the size of the MoM matrix (Equation (4.16)).

In [29], the concept of Macro Basis Functions (MBFs) were introduced. The aim of these MBFs are to subdivide the problem geometry into separate parts for which a macro basis can be generated. The objective of the new macro basis functions is to reduce the number of unknowns, N, associated with the MoM. The CBFM aims to utilise a similar approach, but is more general and also extensively takes the mutual coupling between the created domains into account.

This chapter will first present the formulation of the CBFM followed by a brief discussion on its implementation. Thereafter, the CBFM will be applied to two problems for validation. The parallelisation of the CBFM will then be detailed and tested.

5.1 CBFM Formulation

Consider the geometry in Figure 5.1 which illustrates M domains creating an antenna array. Each domain is discretised into N_i identical basis function in accordance with the MoM RWG triangular patches ($N = M \times N_i$). The MoM matrix equation,

$$[Z][I] = [V], \tag{5.1}$$

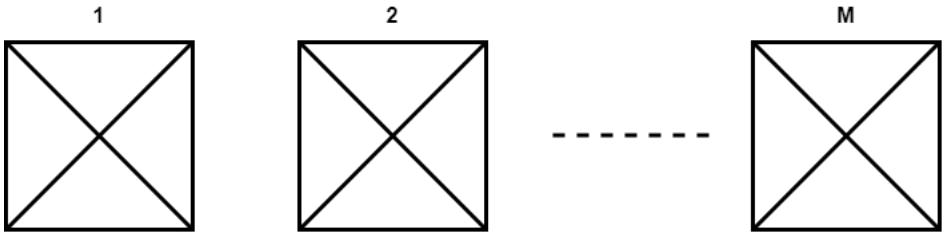


Figure 5.1: A simple array consisting of M domains

can be decomposed in a block like manner as shown in Equation (1.10), repeated here for ease of reading,

$$\begin{bmatrix} [Z_{11}] & [Z_{12}] & \cdots & [Z_{1M}] \\ [Z_{21}] & [Z_{22}] & \cdots & [Z_{2M}] \\ \vdots & \vdots & \ddots & \vdots \\ [Z_{M1}] & \cdots & \cdots & [Z_{MM}] \end{bmatrix} \begin{bmatrix} [I_1] \\ [I_2] \\ \vdots \\ [I_M] \end{bmatrix} = \begin{bmatrix} [V_1] \\ [V_2] \\ \vdots \\ [V_M] \end{bmatrix}, \tag{5.2}$$

where $[Z_{pq}]$ is a sub-matrix of dimension $N_i \times N_i$ and $[I_p]$ and $[V_p]$ are of dimension $N_i \times 1$ for $p, q = 1, ..., M$.

5.1.1 Generating the primary basis functions

From [10], the primary basis functions, $[J_i^{prim}]$ are created to model the self interactions within a specific domain. $[J_p^{prim}]$ is calculated using

$$[Z_{pp}][J_p^{prim}] = [V_p] \text{ for } p = 1, ..., M \tag{5.3}$$

with J_p^{prim} being of dimension $N_i \times 1$. It is important to note that while the problem may be large in terms of N, only a subset N_i is operated on. This allows for a more speedy use of the LU-decomposition.

5.1.2 Generating the secondary basis functions

Following the generation of primary basis functions, [10] introduces secondary basis functions to model the mutual coupling effects between domains. These secondary basis functions are calculated using

$$[Z_{pp}][J_{pq}^{sec}] = -[Z_{pq}][J_q^{prim}] \text{ for } q = 1, 2, ..., p-1, p+1, ..., M, \tag{5.4}$$

where $[J_{pq}^{sec}]$ is of dimension $N_i \times 1$ and is the pth secondary basis function of the qth domain. Following, $[Z_{pq}]$ is the mutual coupling matrix between domains p and q.

5.1.3 Creating the reduced CBFM matrix equation

Currently, for each of the M domains, there is 1 primary basis function and $M-1$ secondary basis functions. These basis functions are then aggregated column wise

$$[J_p^{CBFM}] = [[J_p^{prim}], ..., [J_{pq}^{sec}]] \text{ for } q = 1, 2, ..., p-1, p+1, ..., M, \tag{5.5}$$

keeping in mind for $[J_{pq}^{sec}]$, $p \neq q$. Thus, the dimension of $[J_p^{CBFM}]$ is $N_i \times M$. Now, $[I]$, from Equation (5.1) can be expressed as linear combination of the primary and secondary basis functions as

$$[I] = \sum_{p=1}^{M} \alpha_p^{(1)} \begin{bmatrix} [J_p^{CBFM,(1)}] \\ [0] \\ \vdots \\ [0] \end{bmatrix} + \sum_{p=1}^{M} \alpha_p^{(2)} \begin{bmatrix} [0] \\ [J_p^{CBFM,(2)}] \\ \vdots \\ [0] \end{bmatrix} + ... + \sum_{p=1}^{M} \alpha_p^{(M)} \begin{bmatrix} [0] \\ \vdots \\ [0] \\ [J_p^{CBFM,(M)}] \end{bmatrix} \tag{5.6}$$

where $\alpha_p^{(i)}$ is the unknown complex expansion coefficients and the k in $[J_p^{CBFM,(k)}]$ refers to the kth basis function in the aggregated matrix, i.e. the kth column of matrix $[J_p^{CBFM,(k)}]$. Finally, substituting (5.6) into (5.1) provides the reduced CBFM equation

$$[Z^{CBFM}][I^{CBFM}] = [V^{CBFM}], \tag{5.7}$$

where $[Z^{CBFM}]$ is of dimension $M^2 \times M^2$ and $[I^{CBFM}]$ and $[V^{CBFM}]$ are of dimension $M^2 \times 1$. Solving for $[I^{CBFM}]$ yields the unknown complex coefficients $\alpha_p^{(i)}$.

The reduced matrix $[Z^{CBFM}]$ is generated using $M \times M$ matrix blocks $[Z_{pq}^{CBFM}]$ which are calculated using the inner product operator $\langle , \rangle$ as

$$[Z_{pq}^{CBFM}] = \langle [J_p^{CBFM}]^T, [Z_{pq}][J_q^{CBFM}] \rangle. \tag{5.8}$$

Likewise, $[V^{CBFM}]$ is generated using blocks of $[V_p^{CBFM}]$ as follows,

$$[V_p^{CBFM}] = \langle [J_p^{CBFM}]^T, [V_p] \rangle. \tag{5.9}$$

5.2 Implementing the CBFM

Figure 5.2 shows an overview of the CBFM with each of the steps detailed above. In CEMACS, LAPACK is used for the linear algebra as mentioned in Table 3.1.

Considering the structure of the reduced matrices, a problem quickly becomes apparent. $[Z^{CBFM}]$ is of dimension $M^2 \times M^2$ and if M, the number of basis functions, is large, the exact problem, that of a large LU-decomposition, occurs. This can be solved using Singular Value Decomposition which will be detailed below.

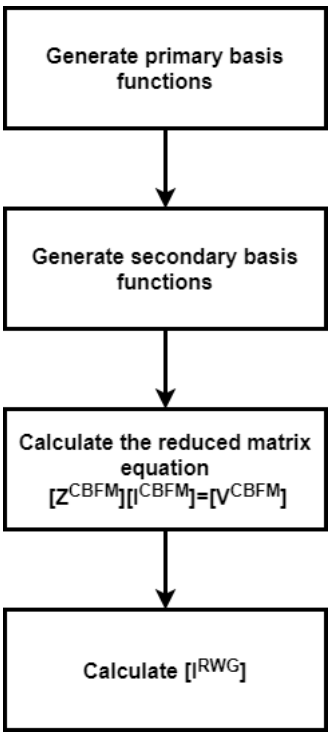


Figure 5.2: A flow diagram illustrating the steps of the CBFM.

5.2.1 Reducing the number of basis functions using Singular Value Decomposition (SVD)

Consider $[J^{CBFM}]$ in Equation (5.5), which consists of the basis functions related to a domain in a column augmented matrix. There is a possibility that the basis functions within are linearly dependant i.e. some basis functions may be redundant due to its small contribution when computing the reduced matrix equation.

The Singular Value Decomposition (SVD) is therefore used to generate a new $[J^{CBFM\_NEW}]$ consisting of a set of orthogonal basis functions. The SVD decomposes $[J^{CBFM}]$ as

$$[J^{CBFM}] = [U][\Sigma][V]^T, \tag{5.10}$$

where $[U]$ and $[V]$ are unitary matrices and $[\Sigma]$ is a diagonal matrix with the entries of the diagonal are the singular values (σ_n) in descending order. A threshold (τ) is selected based on the solution accuracy desired, from [30] typically $10^{-5} < \tau < 10^{-3}$, and the normalised singular values (σ_n/σ_1) exceeding this are discarded.

The columns, n, of $[U]$ corresponding the σ_n kept are then taken to form $[J^{CBFM\_NEW}]$.

5.3 Numerical Results

As mentioned in previously in Section 4.3, it is imperative to validate the results of CEMACS. For consistency, the same examples as in the MoM will be used.

5.3.1 Applying CEMACS CBFM to a square PEC plate array

Consider again the square plate array in Figure 4.4 consisting of one hundred square plates evaluated at 300 MHz. The E-field cut at the red line is compared to FEKO in Figure 5.3 and has an $\epsilon = 6.39\%$. This result is identical to CEMACS MoM, where the inconsistencies are a result of near singularities, and validates the implementation of the CEMACS CBFM solver.

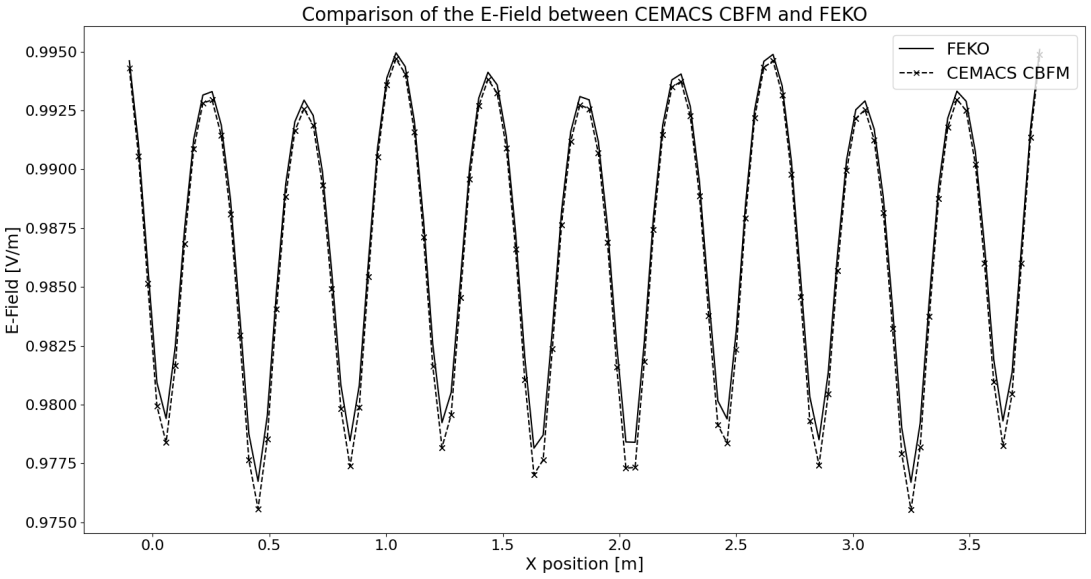


Figure 5.3: A comparison of the E-Field between CEMACS CBFM and FEKO for a plate array.

5.3.2 Applying CEMACS CBFM to a Vivaldi array

Consider again the array of three Vivaldi antennas in Figure 4.6. Applying the CEMACS CBFM to this geometry yields the E-plane gain at 500 MHz in Figure 5.4 and the magnitude and phase of S_{11} for the frequency range of 500 MHz to 1 GHz as shown in Figures 5.5 and 5.6 respectively.

Table 5.1 presents the error between CEMACS CBFM and FEKO, which is low and within limits of acceptability, with improvements needed in the area

of near singularity treatment. The validation of the CEMACS CBFM is now proven and will be accelerated in the next section.

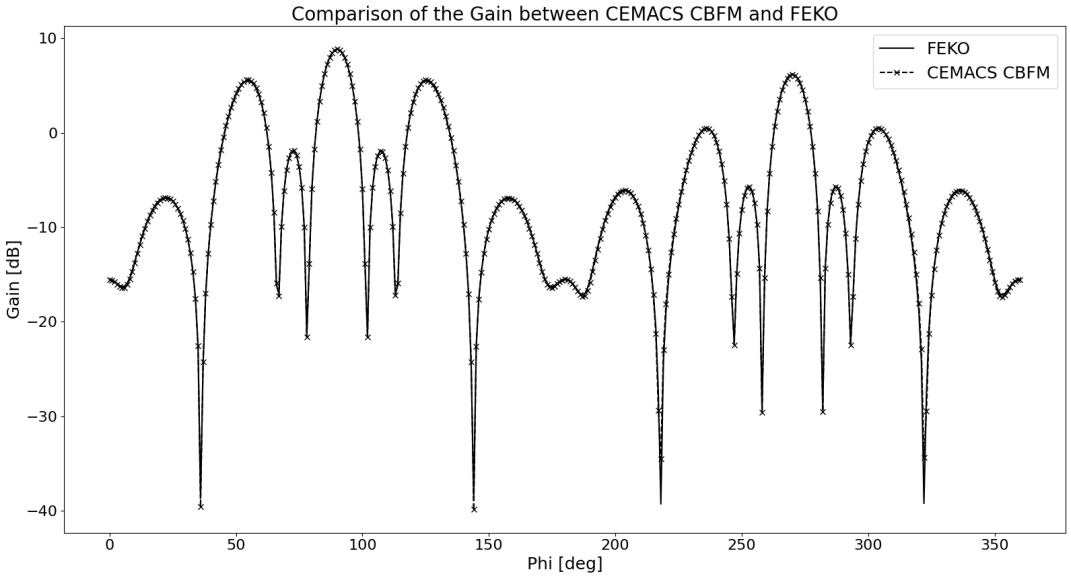


Figure 5.4: A comparison of the E-plane gain (dB) between CEMACS CBFM and FEKO for a Vivaldi antenna array.

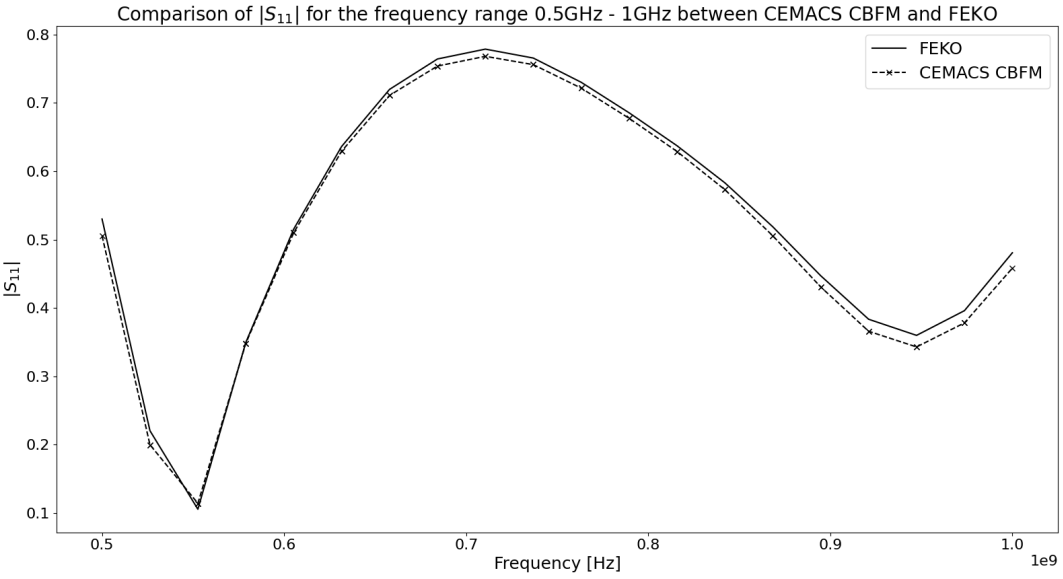


Figure 5.5: A comparison of $|S_{11}|$ between CEMACS CBFM and FEKO for a Vivaldi antenna array.

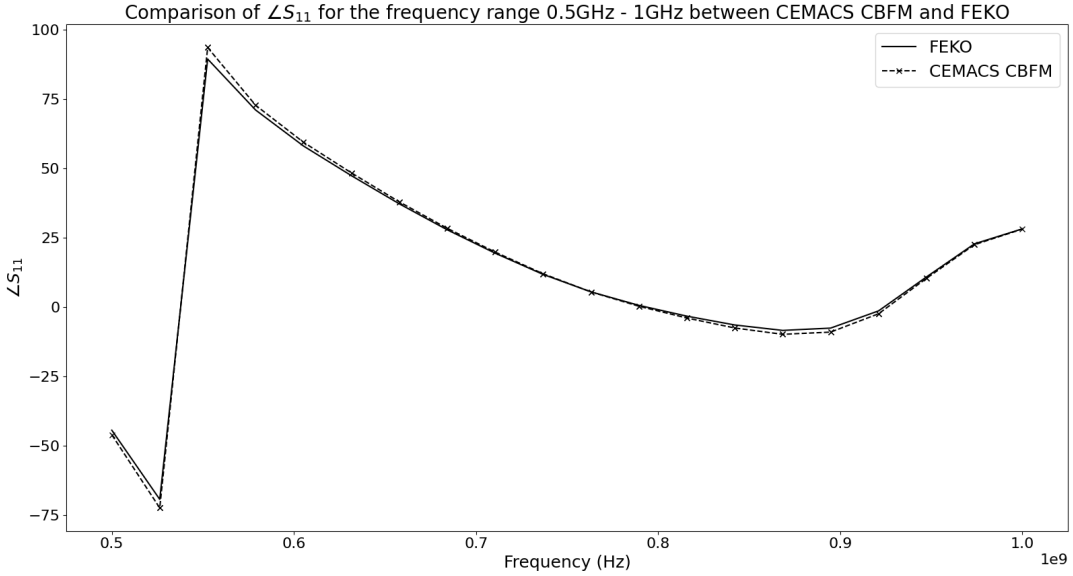


Figure 5.6: A comparison of the phase of S_{11} between CEMACS CBFM and FEKO for a Vivaldi antenna array.

Calculated Quantity	$\epsilon_{\%}$
E-plane gain	3.84%
$\lvert S_{11}\rvert$	2.29%
$\angle S_{11}$	3.86%

Table 5.1: Relative error percentages ($\epsilon_{\%}$) between CEMACS CBFM and FEKO for the Vivaldi array.

5.4 Parallelisation of the CBFM

Following the validation of the accuracy of CEMACS CBFM, its parallelisation, using the tools presented in Chapter 2, will now be discussed. To obtain the MPI and hybrid MPI-OpenMP results, a single node consisting of 24 cores was used with the specifications found in Appendix A and the results when testing the CUDA GPU implementation were obtained using a Tesla V100 GPU with 5120 CUDA cores (specifications in Appendix B), and an Intel Xeon CPU E5-2690 @ 2.60GHz processor.

5.4.1 MPI

As discussed in Section 3.2.1, the matrix equation 5.2 is spread by rows as equally as possible among the MPI processes. Consider the simplest example with two domains ($M = 2$) and two MPI processes (($P = 2$). Each process will calculate all blocks of $[Z]$ and $[V]$ on their row. Process 0, for example,

will have

$$[Z_{11}][Z_{12}][I_1] = [V_1] \tag{5.11}$$

with $[I_1]$ being empty. The primary basis functions will then be calculated, again for Process 0,

$$[Z_{11}][J_1^{\text{prim}}] = [V_1] \tag{5.12}$$

using LU-decomposition. Now, to calculate the secondary basis function required for domain 1, the domain on Process 0,

$$[Z_{11}][J_{12}^{\text{sec}}] = -[Z_{12}][J_2^{\text{prim}}] \tag{5.13}$$

is used. $[J_2^{\text{prim}}]$ is however on Process 1 so an MPI communication call is required to transmit the primary basis functions between the processes. Following, $[J_1^{\text{CBFM}}]$ from Equation 5.5 is formulated.

When calculating the reduced matrix equation all the necessary data for $[V_1^{\text{CBFM}}]$ is already on the required process. However, $[Z_{pq}^{\text{CBFM}}]$ requires the communication of $[J_q^{\text{CBFM}}]$. In the case of Process 0,

$$[Z_{12}^{CBFM}] = \langle [J_1^{CBFM}]^T, [Z_{12}][J_2^{CBFM}] \rangle. \tag{5.14}$$

$[J_2^{CBFM}]$ is required from Process 1. Thus, another MPI communication is required to broadcast this. After the reduced blocks of $[Z^{CBFM}]$ and $[V^{CBFM}]$ are calculated, they need to be communicated to the root process, typically Process 0. $[I^{\text{CBFM}}]$ is then calculated and thereafter the final solution $[I]$. This example, illustrated in Figure 5.7, scales to M domains on P processes.

The important parts to note is the gathering of all primary basis functions ($[J^{\text{PRIM}}]'s$) and the concatenated basis function matrices ($[J^{\text{CBFM}}]'s$) on each of the processes. This forms the bulk of the communication overhead i.e. the gathering of the final solution $[I]$ is trivial in comparison, $N_i \times M + N_i \times M^2$ versus N, recalling that $N = N_i \times M$.

Due to efficiency and practicality, it follows that a process will incur no penalty for self sending. This means that the more rows of the initial matrix equation a process has, the less the effect of communication, with the new data transmission formula as,

$$\text{Data Transferred between processes} = N_i \times (M - n) + N_i \times (M - n)^2 \tag{5.15}$$

where n is the number of rows per process. This can clearly be seen in the data collected in Figure 5.9 where the MPI implementation of the CBFM is applied to the 24 element bow tie array shown in Figure 5.8 with variable amounts of unknowns (N). Looking at a single problem size, it is clearly illustrated that increase in speed-up starts to decline as the number of processes increase. This is consistent with Equation (5.15). Now, looking at the graph as a whole, the larger N is, the higher the speed-up. This is explained due to the ratio between actual parallel computation versus communication, which increases with problem size.

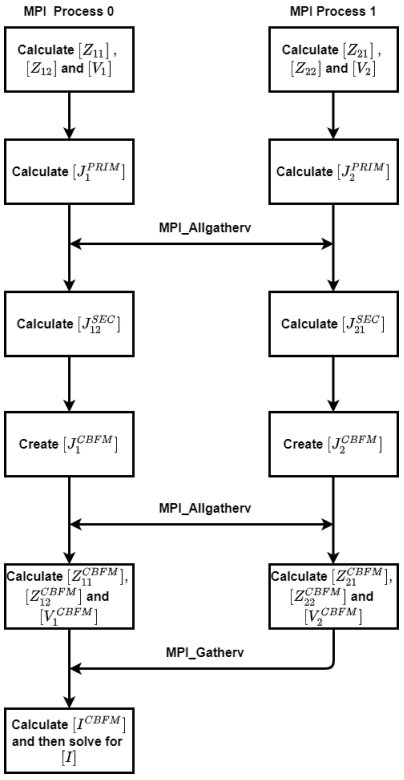


Figure 5.7: Parallelisation of the CBFM using 2 domains and 2 MPI processes.

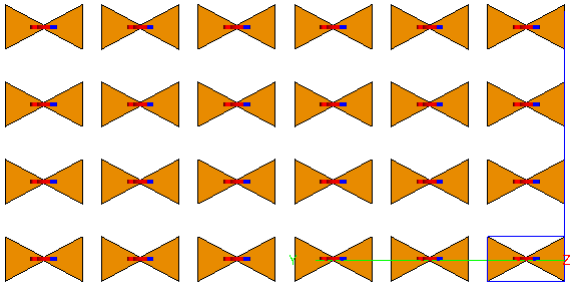


Figure 5.8: An array of twenty four bow tie antennas.

Now, considering an array of square plates similar to Figure 4.4 where $N = 12480$ but the number of domains, M, is variable, the speed-up is equal at lower process numbers but starts to widen as the processes increase. This is illustrated in Figure 5.10, and in contrast to Figure 5.9, where larger is better, the smaller M is, the greater the speed-up. This is attributed to the size of the reduced CBFM matrix blocks $[Z_{pq}^{CBFM}]$ and $[V_p^{CBFM}]$, which scale with M as $\mathcal{O}(M^2)$.

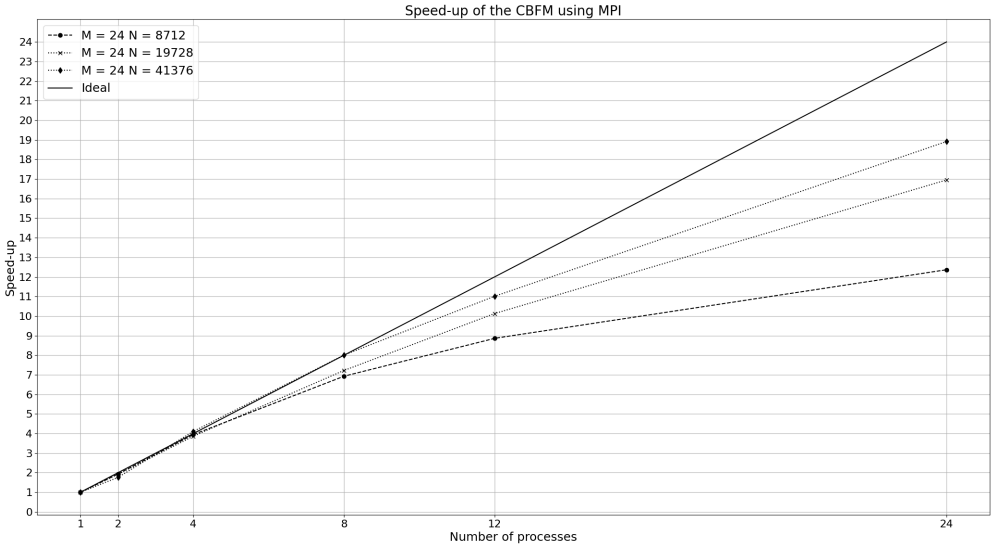


Figure 5.9: Speed-up of the CBFM for a bow tie array with variable N

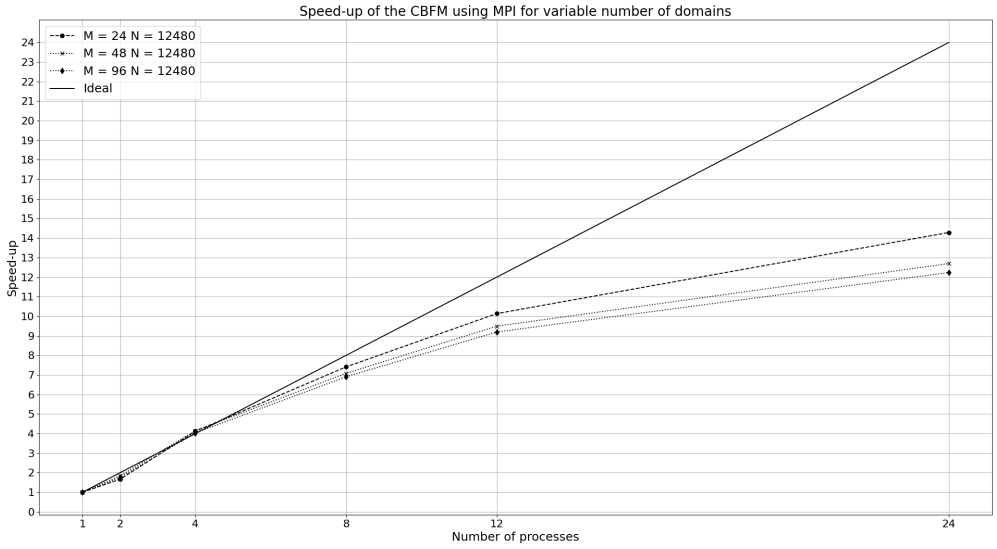


Figure 5.10: Speed-up of the CBFM for a square plate array with variable M

5.4.2 Hybrid MPI-OpenMP

Consider again the bow-tie array in Figure 5.8, with the results of applying the hybrid MPI-OpenMP technique shown in Figure 5.11. OpenMP is applied to the filling of the blocks of $[Z]$, as well as to the linear algebra routines using OpenBLAS. The test was completed on a single node consisting of two processors (12 cores per processor) to illustrate the effects of over threading as described in Section 2.2.4. In the case of 8 processes being used, the speed-up when using three threads is greater than that of four threads which is consistent to what was mentioned previously.

Generally, the speed-up was satisfactory and concretely illustrates the importance of choosing the correct resources in order to complete the task most effectively. Consider the speed-up of 24 MPI processes using 24 cores for $N = 41376$ in Figure 5.9, which is 19× compared to 8 processes and 3 threads which provides a speed-up of 23.7×. This large speed-up can be attributed to the threaded linear algebra routines used to calculate $[I^{CBFM}]$ and $[I]$, which when using MPI, are calculated serially. The same hardware resources were used, but the result is markedly different.

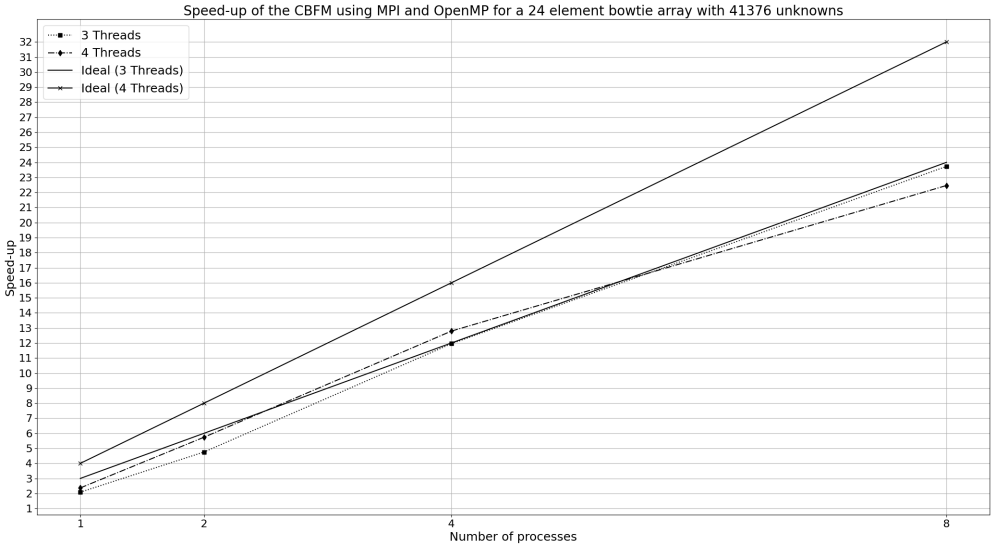


Figure 5.11: Hybrid MPI-OpenMP speed-up using the CBFM on a 24 element bow-tie array.

5.4.3 Implementing the CBFM on a GPU using CUDA

The implementation of the CBFM using CUDA is simpler compared to the MPI implementation. Due to the nature of the CBFM, where a row in the block matrix equation (Equation 5.2) cannot be solved independently, the most effective implementation is the generation of the complete $[Z]$ matrix on the GPU. Naturally, this means that problem size is constrained by available GPU memory. It is not practical to keep transferring the necessary blocks of $[Z]$ needed due to the cost of data transference (Section 2.2.5). A solution to this, but not in the scope of this work, is the usage of an array of multiple GPU's. This naturally comes with its own challenges and needs to be investigated thoroughly. Apart from the filling of $[Z]$ and $[V]$, which are completed using CUDA kernels written by the author, all linear algebra is completed using available libraries as mentioned in Section 3.1.3.1. In terms of accuracy, the results obtained are identical to those shown in Section 5.3.

When applied to a bow-tie array as in Figure 5.8, consisting of $M = 12$ elements the speed-up of the CUDA implementation (using the grid size in Section 3.2.2) compared to the serial CBFM implementation on the CPU is illustrated in Figure 5.12. With no large LU-decomposition, the runtime is

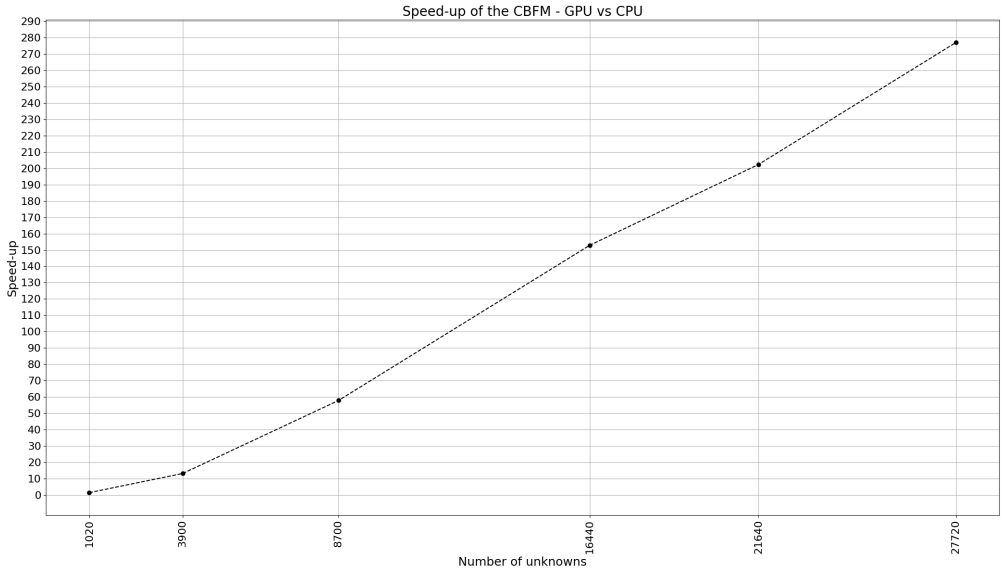


Figure 5.12: The speed-up of the CBFM implementation using CUDA on a GPU (Tesla V100) compared to the serial CBFM implementation using a CPU (Intel Xeon CPU E5-2690 @ 2.60GHz)

dictated by the fill speed of $[Z]$, a strong point of the GPU. Naturally, as problem size gets larger, more of the GPU gets used and the speed-up increases.

5.5 Conclusion

This chapter presented the formulation and implementation of the CBFM. Following this, the CBFM was successfully validated, with an error of 6.39%, against FEKO. Parallelisation using MPI and OpenMP was detailed and tested yielding positive results. Finally, the CBFM was implemented on a GPU to great success, yielding accurate results and an acceptable speed-up. The next chapter will discuss the Domain Green's Function Method (DGFM) another domain decomposition method.

Chapter 6

Domain Greens Function Method

Continuing from the CBFM discussed in the previous chapter, the Domain Green's Function Method (DGFM) is another domain decomposition technique. As presented in [11], the DGFM is a perturbation technique for large disjoint antenna arrays which formulates an active impedance matrix in order to account for mutual coupling. Compared to the CBFM no basis functions are calculated which leads to an increase in speed and a reduction of memory. The DGFM, however, is less accurate than the CBFM when there is strong mutual coupling present in the array. A solution for this is presented in the next chapter.

This chapter will begin with the formulation of the DGFM followed by a brief discussion on its implementation. Following this the DGFM will be validated using two examples. Subsequently, the DGFM will be parallelised and its scaling will be evaluated.

6.1 DGFM Formulation

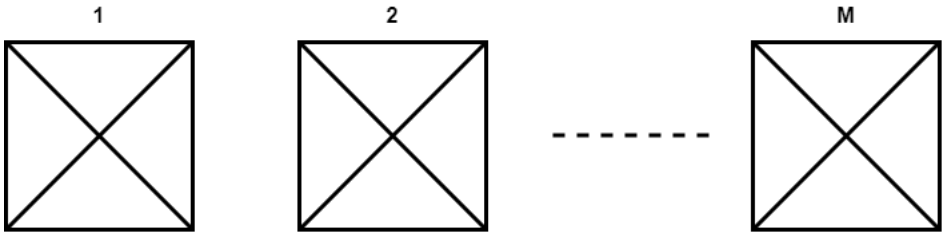


Figure 6.1: A simple array consisting of M domains

Consider the geometry in Figure 6.1 which illustrates M domains creating an antenna array. Each domain is discretised into N_i identical basis function in accordance with the MoM RWG triangular patches ($N = M \times N_i$). The

MoM matrix equation

$$[Z][I] = [V] \tag{6.1}$$

can be decomposed in a block like manner

$$\begin{bmatrix} [Z_{11}] & [Z_{12}] & \cdots & [Z_{1M}] \\ [Z_{21}] & [Z_{22}] & \cdots & [Z_{2M}] \\ \vdots & \vdots & \ddots & \vdots \\ [Z_{M1}] & \cdots & \cdots & [Z_{MM}] \end{bmatrix} \begin{bmatrix} [I_1] \\ [I_2] \\ \vdots \\ [I_M] \end{bmatrix} = \begin{bmatrix} [V_1] \\ [V_2] \\ \vdots \\ [V_M] \end{bmatrix}, \tag{6.2}$$

where $[Z_{pq}]$ is a sub-matrix of dimension $N_i \times N_i$ and $[I_p]$ and $[V_p]$ are of dimension $N_i \times 1$ for $p, q = 1, ..., M$.

As stated in [11], the assumption that $[I_p]$, the surface current density on domain p is just a scaled version of the current on any other domain, is central to the formulation of the DGFM. Using this assumption, the current on domain $p = 1$ ($[I_1]$) can be calculated by equating the currents on other domains ($q = 2, ..., M$) to a scaled $[I_1]$ as shown,

$$[I_2] = \alpha_{21}[I_1] \quad [I_3] = \alpha_{31}[I_1] \quad [I_M] = \alpha_{M1}[I_1] \tag{6.3}$$

with

$$\alpha_{ij} = \frac{C_i}{C_j}, \text{ for } i, j = 1 \dots M. \tag{6.4}$$

The scaling factor, $\frac{C_i}{C_j}$, is the ratio of complex excitation coefficients with respect to their specific domains. Substituting Equation (6.3) into Equation (6.1), $[Z_1^{ACT}]$, the active impedance matrix for domain $p = 1$ is calculated as

$$\begin{aligned} [Z_1^{\text{act}}] &= \left[[Z_{11}] + \alpha_{21}[Z_{12}] + \cdots + \alpha_{M1}[Z_{1M}] \right] \\ &= \left[\sum_{m=1}^{M} \alpha_{m1}[Z_{1m}] \right]. \end{aligned} \tag{6.5}$$

$[I_1]$ can then be solved using LU-decomposition for the following reduced matrix equation

$$[Z_1^{\text{act}}][I_1] = [V_1]. \tag{6.6}$$

The complete surface current, $[I]$, from Equation (6.1) can then be obtained by repeating the above approach for the other domains $p = 2, ..., M$.

6.2 Implementing the DGFM

Shown in Figure 6.2, is an overview of the DGFM formulation as discussed above. The filling and solving of $[Z]$ and $[I]$ respectively are simple and have been discussed in previous chapters. Of note are the weights, α, discussed below.

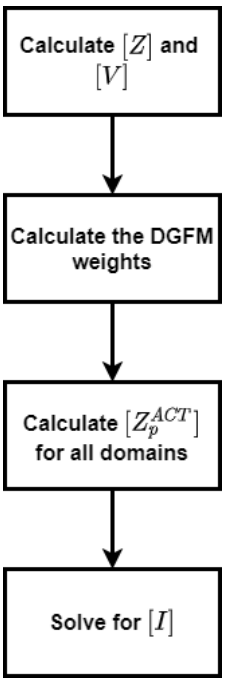


Figure 6.2: The steps required to compute the DGFM.

6.2.1 DGFM weights for an edge feed

When concerning edge feeds, the weights (α_{qp}) are

$$\alpha_{qp} = \sum_{0}^{N_i} \frac{[V_q]}{[V_p]} \tag{6.7}$$

where the length of $[V_p]$ is N_i. Here, V_p and V_q, are the complex port excitations associated with domains p and q respectively.

6.2.2 DGFM weights for a plane wave

If a plane wave is incident on an array at an angle, it follows that array elements (domains) will be progressively illuminated, with the induced phase dependant on the angle of incidence. The weights α_{qp} will then be

$$\alpha_{qp} = e^{-j\vec{k}\cdot\vec{r}}, \tag{6.8}$$

where $\vec{k}$ is the propagation vector and $\vec{r}$ is the vector pointing from the centre of domain q to the centre of domain p.

6.3 Numerical Results

As mentioned in previously in Section 4.3, it is imperative to validate the results of CEMACS. For consistency, the same examples as in the MoM will

be used.

6.3.1 Applying CEMACS DGFM to a square PEC plate array

Consider again the square plate array in Figure 4.4 consisting of one hundred square plates at 300 MHz. The E-field cut at the red line is given in Figure 6.3 and has an ϵ compared to FEKO of 6.39%. This result is identical to CEMACS MoM, where the inconsistencies are a result of near singularities, and validates the implementation of the DGFM solver.

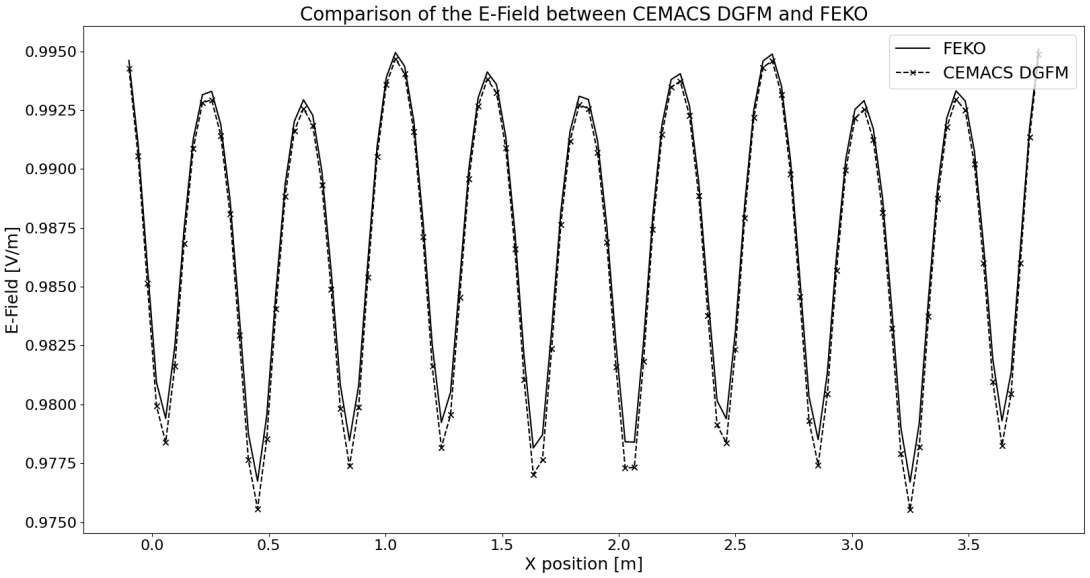


Figure 6.3: A comparison of a square plate array's E-Field between CEMACS DGFM and FEKO.

6.3.2 Applying CEMACS DGFM to a Vivaldi array

Now, CEMACS DGFM will be applied to the three element Vivaldi array in Figure 4.6. The parameters, as mentioned in the case of the MoM and CBFM, are the E-plane gain and the magnitude and phase of S_{11} with the error compared to FEKO shown in Table 6.1. The high error of 12.96% of the E-Plane gain, shown in Figure 6.4, can be attributed to strong mutual coupling between the elements in the vivaldi array. Comparatively, the error percentage for the magnitude and phase of S_{11}, Figures 6.5 and 6.6 respectively, are low and consistent to the results obtained for CEMACS MoM and CBFM.

Calculated Quantity	$\epsilon_{\%}$
E-plane gain	12.96%
$\lvert S_{11}\rvert$	2.26%
$\angle S_{11}$	4.90%

Table 6.1: Relative error percentages ($\epsilon_{\%}$) between CEMACS MoM and FEKO for three calculated quantities.

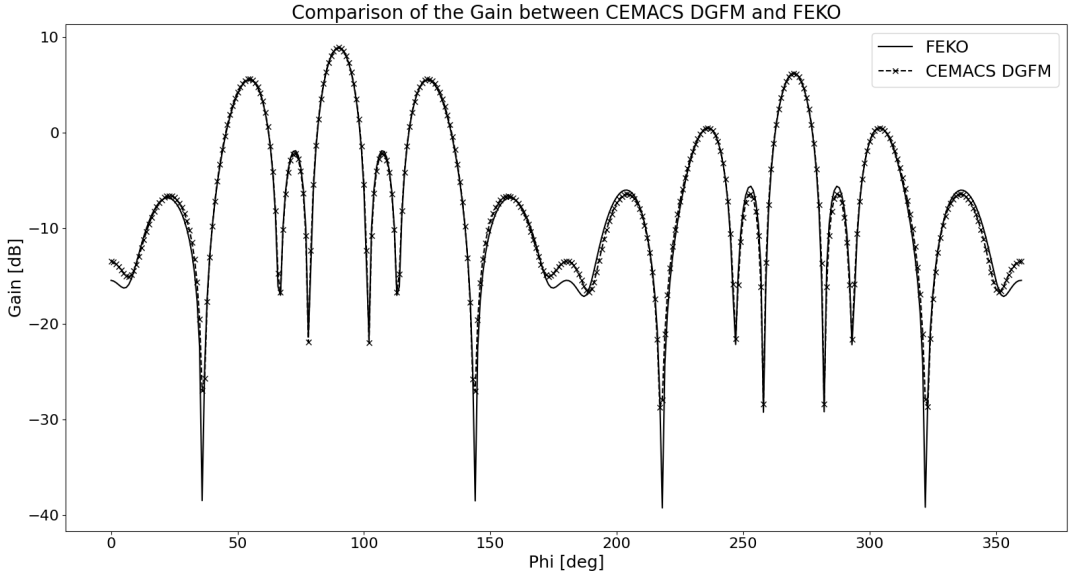


Figure 6.4: A comparison of the E-plane gain (dB) between CEMACS DGFM and FEKO for a Vivaldi antenna array.

6.4 Parallelisation of the DGFM

Following the validation of the accuracy of CEMACS DGFM, its parallelisation will now be discussed. The parallelisation of the DGFM, using MPI, a hybrid MPI-OpenMP approach and CUDA will be discussed below. To obtain the MPI and hybrid MPI-OpenMP results, a single node consisting of 24 cores was used with the specifications found in Appendix A and the results when testing the CUDA GPU implementation were obtained using a Tesla V100 GPU with 5120 CUDA cores (specifications in Appendix B), and an Intel Xeon CPU E5-2690 @ 2.60GHz processor.

6.4.1 MPI

The DFGM lends itself well to MPI parallelisation due to the independence of the calculation of $[Z_p^{ACT}]$. In the case where there are edge feeds present, the weights require the $[V_q]$'s from other processes. It is faster to calculate the necessary $[V_q]$'s than to communicate them between processes which results in

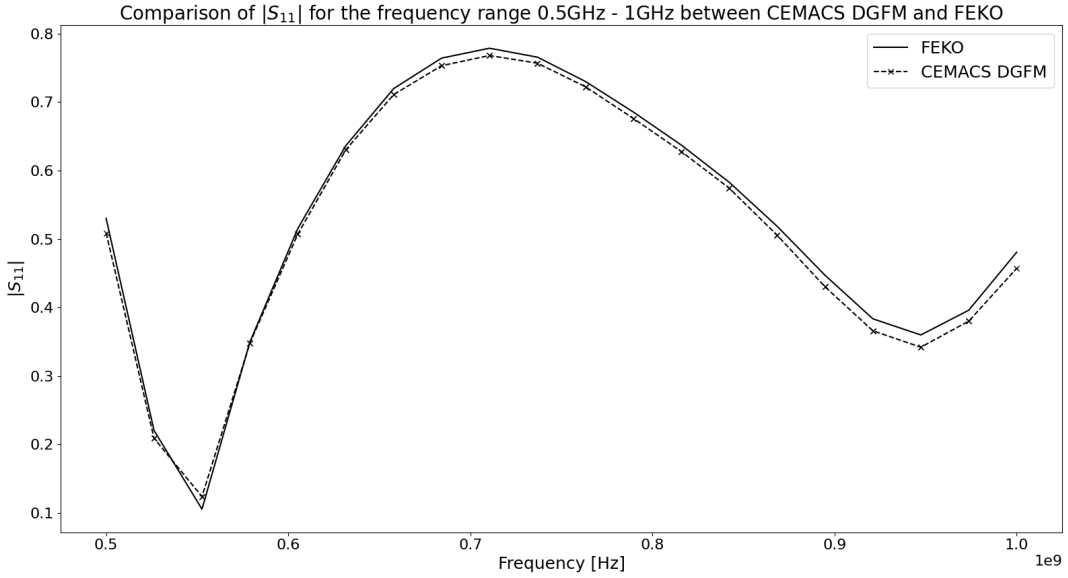


Figure 6.5: A comparison of $|S_{11}|$ between CEMACS DGFM and FEKO for a Vivaldi antenna array.

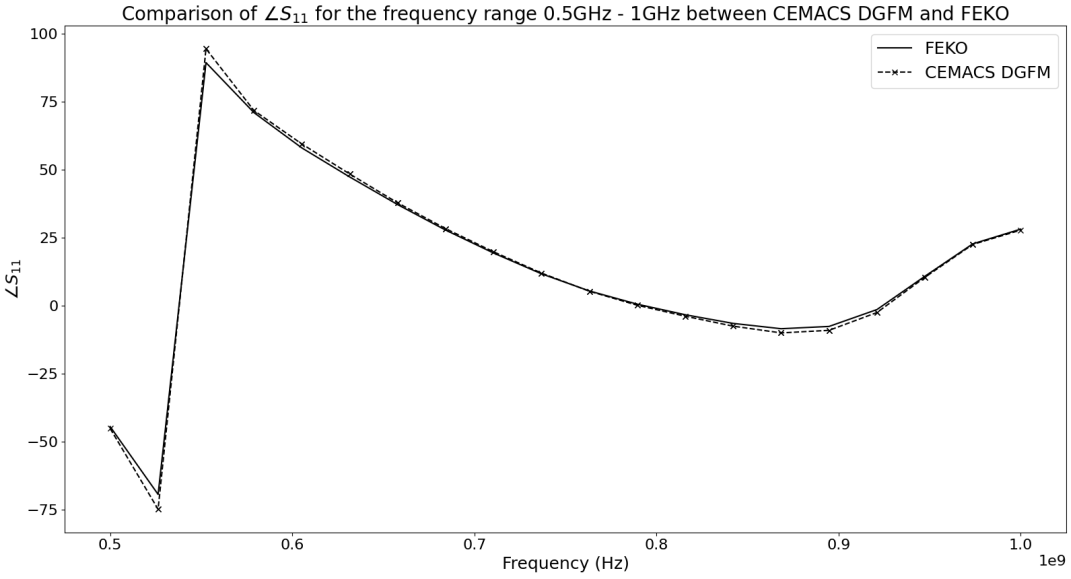


Figure 6.6: A comparison of the phase of S_{11} between CEMACS DGFM and FEKO for a Vivaldi antenna array.

each process calculating the entire $[V]$. A simple example with two processes and two domains is illustrated in Figure 6.7. Figure 6.8 shows the results when applying this method to the 24 element bow-tie array in Figure 5.8 . The minimal interprocess communication makes the speed-up less than ideal as the number of processes get larger, but the rate of decline is gradual. When looking at the different problem sizes, the speed-up's are quite close, which is

due to the cost of managing the processes having a greater influence compared the cost of the actual data being communicated.

Then, considering the bow-tie array with a constant N and variable M, the speed-up is closely grouped as shown in Figure 6.9. $M = 24$ at 24 processes does show a slight speed-up increase, attributed to the calculation of the DGFM weights, as the weights are are of $\mathcal{O}(M)$.

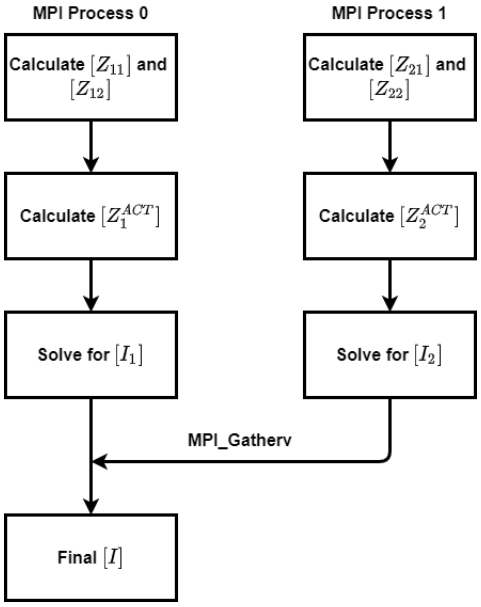


Figure 6.7: MPI parallelisation of the DGFM for 2 domains using 2 MPI processes.

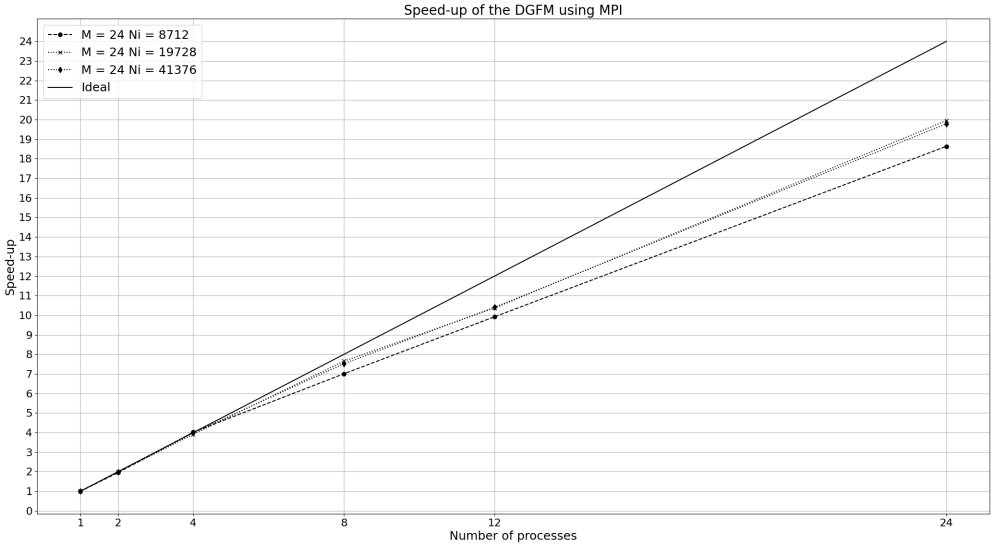


Figure 6.8: Speed-up of the DGFM using MPI for a bow-tie array with variable N

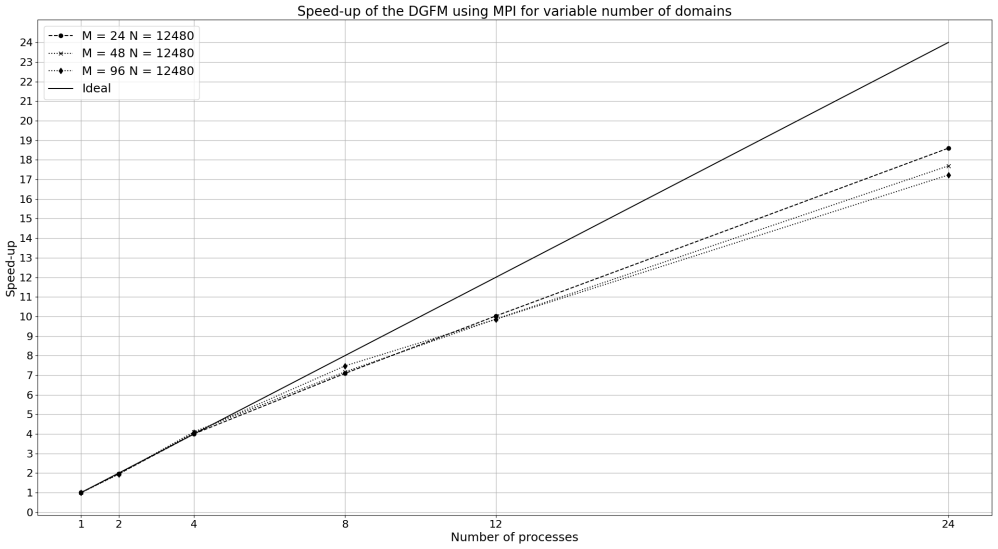


Figure 6.9: Speed-up of the DGFM using MPI for a bow-tie array with variable M

6.4.2 Hybrid OpenMP-MPI

Considering again the bow-tie array in Figure 5.8, the results of the application of the hybrid MPI-OpenMP technique is shown in Figure 6.10. As in Section 5.4.2, threading was applied to the filling of the blocks of $[Z]$ and $[V]$ as well as the linear algebra routines. In addition, the calculation of the DGFM weights were also parallelised. The results shown in Figure 6.10 also shows the effect of thread overallocation as seen in Figure 5.11, which is discussed in Section 5.4.2 albeit to a lesser extent. The speed-up is overall less than that obtained from the CBFM implementation but is still satisfactory.

6.4.3 Implementing the DGFM on a GPU using CUDA

Implementing the DGFM using CUDA is different compared to the CBFM in the sense that the rows of the block form of $[Z]$ are independent and need not be stored in memory for a solution. For problem sizes less than the available GPU memory, it is faster to compute $[Z]$ in its entirety rather than row by row. However, there is the option for row filling in the case of a problem exceeding the available memory. Both the filling of $[Z]$ and the calculation of the DGFM weights are completed using a CUDA kernel written by the author. As in the case of the CBFM, linear algebra routines are completed using a GPU specific library as shown in Table 3.1. In terms of accuracy, the results obtained are identical to those shown in Section 6.3.

The results of applying this to a 12 element bow-tie array, similar to that shown in Figure 5.8 (12 elements instead of 24 elements), is shown in Figure 6.11, where the CUDA implementation on a GPU (using the grid size in Section 3.2.2) is compared to the serial implementation of the DGFM on a

CPU. The large speed-up, is attributed to the considerably few linear algebra routines required. The filling of $[Z]$ and the calculation of the weights are considerably more independent, in terms of parallelism, than than that of some linear algebra routines. Also to be considered is that a single CUDA kernel fills the entire $[Z]$ while each linear algebra routine calls a new kernel.

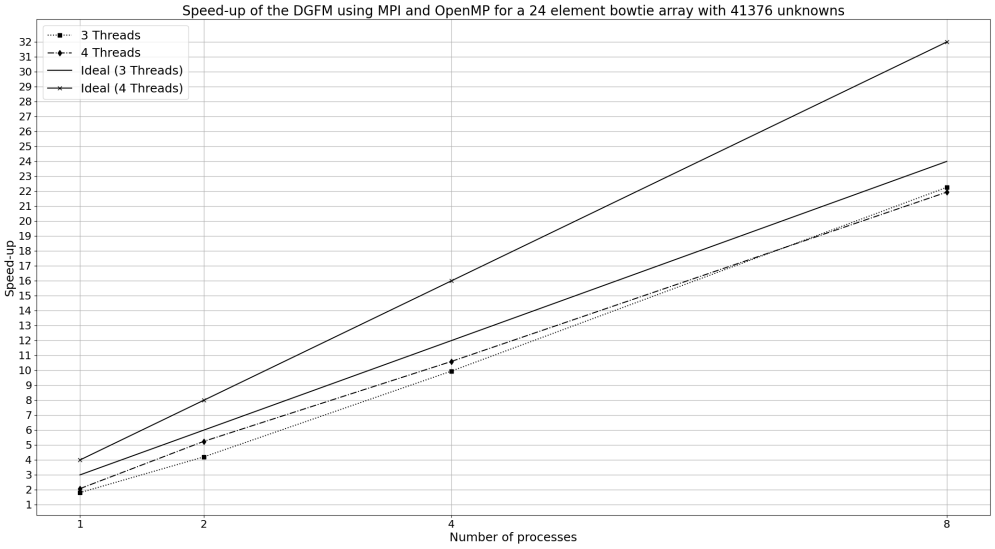


Figure 6.10: Hybrid MPI-OpenMP speed-up using the DGFM on a 24 element bow-tie array.

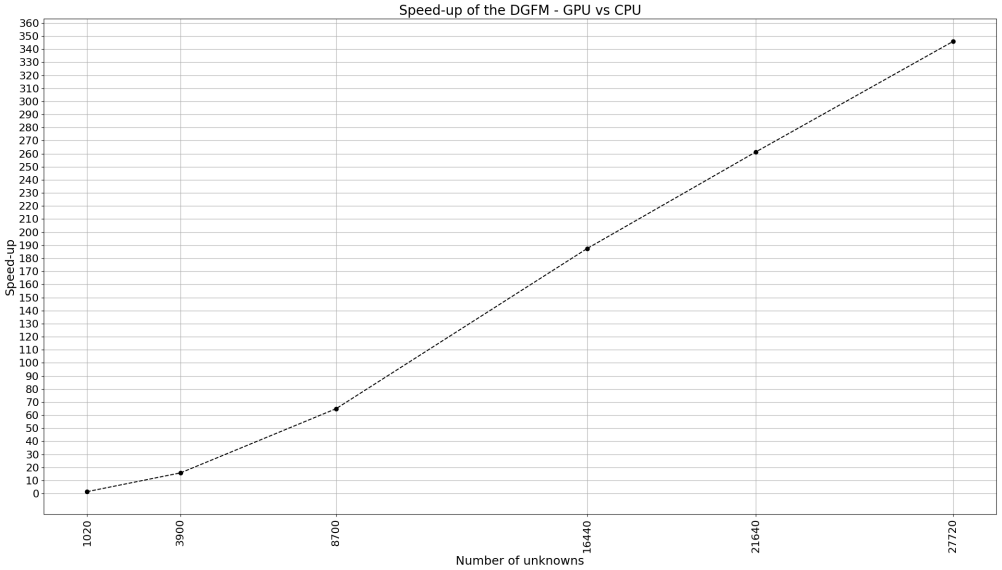


Figure 6.11: The speed-up of the DGFM implementation using CUDA on a GPU (Tesla V100) compared to a serial DGFM implementation on a CPU (Intel Xeon CPU E5-2690 @ 2.60GHz)

6.5 Conclusion

In this chapter the DGFM was formulated, implemented and validated successfully. The DGFM was then successfully parallelised using MPI and OpenMP. Finally, the DGFM was implemented on a GPU to great success, achieving a speed-up of 350× compared to a serial CPU DGFM implementation at the largest problem size. In the next chapter, the final domain decomposition technique, one meant to address the problem the DGFM has with mutual coupling, the Improved Domain Green's Function Method (i-DGFM), will be presented.

Chapter 7

Improved Domain Greens Function Method

Mentioned in Chapter 6 is that the accuracy of the DGFM is low in cases of strong mutual coupling. The Improved Domain Green's Function Method (i-DGFM) discussed in [12] aims to negate this by use of a weight matrix rather than the weight vector as presented in Equation (6.4).

Following the weight matrix formulation, the i-DGFM will be implemented and applied to two problems. The i-DGFM will then be parallelised using MPI and OpenMP followed by its parallelisation on a GPU.

7.1 i-DGFM Formulation

Consider the MoM matrix equation $[Z][J] = [V]$ (J is substituted for I in this case due to the introduction of the identity matrix), for a geometry consisting of M domains, the block decomposition of $[Z]$ is

$$\begin{bmatrix} [Z_{11}] & [Z_{12}] & \cdots & [Z_{1M}] \\ [Z_{21}] & [Z_{22}] & \cdots & [Z_{2M}] \\ \vdots & \vdots & \ddots & \vdots \\ [Z_{M1}] & \cdots & \cdots & [Z_{MM}] \end{bmatrix}, \tag{7.1}$$

which can also be written, as noted in [12], in terms of on and off diagonal matrices as

$$[Z] = [Z_{on}] + [Z_{off}] = \begin{bmatrix} [Z_{11}] & & & \\ & [Z_{22}] & & \\ & & \ddots & \\ & & & [Z_{MM}] \end{bmatrix} + \begin{bmatrix} & [Z_{12}] & \cdots & [Z_{1M}] \\ [Z_{21}] & & \cdots & [Z_{2M}] \\ \vdots & \vdots & & \vdots \\ [Z_{M1}] & \cdots & \cdots & \end{bmatrix}. \tag{7.2}$$

By substituting Equation (7.2) into the MoM matrix equation and multiplying each side with $[Z_{on}^{-1}]$,

$$\begin{aligned} [Z][J] &= [V] \\ ([Z_{on}^{-1}][Z_{on}] + [Z_{on}^{-1}][Z_{off}])[J] &= [Z_{on}^{-1}][V] \\ ([I] + [Z_{on}^{-1}][Z_{off}])[J] &= [J_0] \\ [J] &= ([I] + [Z_{on}^{-1}][Z_{off}])^{-1}[J_0], \end{aligned} \tag{7.3}$$

where $[I]$ is the identity matrix and $J_0 = [Z_{on}^{-1}][V]$ is current on the geometry ignoring coupling. Using the infinite geometric series,

$$(1-r)^{-1} = \sum_{n=0}^{\inf} r^n, \tag{7.4}$$

$([I]+[Z_{on}^{-1}][Z_{off}])^{-1}$ can be rewritten and substituted into Equation (7.3) giving

$$\begin{aligned} [J] &= ([I] + [Z_{on}^{-1}][Z_{off}])^{-1}[J_0] \\ [J] &= \sum_{n=0}^{\inf} (-[Z_{on}^{-1}][Z_{off}])^n [J_0], \end{aligned} \tag{7.5}$$

which converges, as noted in [12], when the magnitude of the eigenvalue of the principal eigenvector of $[Z_{on}^{-1}][Z_{off}]$ is less than one. Substituting the block matrix in Equation (7.1) provides the solution for the MoM currents $[J]$ which in block form is

$$\begin{bmatrix} [J_1] \\ [J_2] \\ \vdots \\ [J_M] \end{bmatrix} = \begin{bmatrix} [J_{01}] - \sum_{m=1,m\neq 1}^{M} [Z_{11}^{-1}][Z_{1m}][J_{0m}] + \sum_{m=1,m\neq 1}^{M} ([Z_{11}^{-1}][Z_{1m}])^2[J_{0m}] - \dots \\ [J_{02}] - \sum_{m=1,m\neq 2}^{M} [Z_{22}^{-1}][Z_{2m}][J_{0m}] + \sum_{m=1,m\neq 2}^{M} ([Z_{22}^{-1}][Z_{2m}])^2[J_{0m}] - \dots \\ \vdots \\ [J_{0M}] - \sum_{m=1,m\neq M}^{M} [Z_{MM}^{-1}][Z_{Mm}][J_{0m}] + \sum_{m=1,m\neq M}^{M} ([Z_{MM}^{-1}][Z_{Mm}])^2[J_{0m}] - \dots \end{bmatrix}, \tag{7.6}$$

where $[J_{0p}]$ is the primary basis function as described in Equation (5.3) and $\sum_{m=1,m\neq 1}^{M} [Z_{11}^{-1}][Z_{1m}][J_{0m}]$ is the sum of the secondary basis functions described in Equation (5.4). In simple terms, Equation (7.6) states that the current on a domain is the sum of its basis functions (primary, secondary, tertiary, etc.).

Limiting Equation (7.6) at secondary coupling (not including greater than secondary basis functions) the currents on a domain p is

$$[J_p] = [J_{0p}] - \sum_{m=1,m\neq p}^{M} [Z_{pp}^{-1}][Z_{pm}][J_{0m}]. \tag{7.7}$$

$[\alpha_{qp}]$ is then $\frac{[J_q]}{[J_p]}$ augmented row-wise as

$$[\alpha_{qp}] = \begin{bmatrix} J_q^1/J_p^1 & J_q^2/J_p^2 & \cdots & J_q^M/J_p^M \\ J_q^1/J_p^1 & J_q^2/J_p^2 & \cdots & J_q^M/J_p^M \\ \vdots & \vdots & \ddots & \vdots \\ J_q^1/J_p^1 & J_q^2/J_p^2 & \cdots & J_q^M/J_p^M \end{bmatrix}, \tag{7.8}$$

with J_q^n and J_p^n the elements of $[J_q]$ and $[J_p]$ for $n = 1, ..., N_i$, N_i the number of RWG basis functions on a domain.

7.2 Implementing the i-DGFM

Implementing the i-DGFM is identical to the CBFM in Chapter 5 for the first three steps shown in Figure 7.1. Following, the weights are calculated according to the method explained above and $[Z_p^{ACT}]$ is calculated using element wise matrix-matrix multiplication. $[I]$ is then finally solved using LU-decomposition on $[Z_p^{ACT}][I_p] = [V_p]$ for all domains.

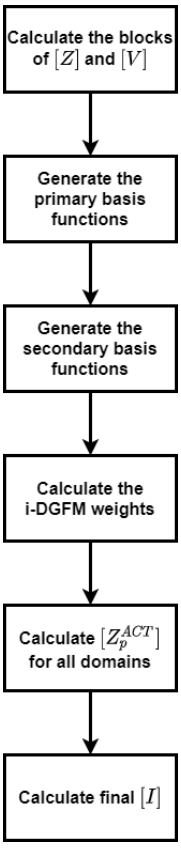


Figure 7.1: The steps required to compute the i-DGFM.

7.3 Numerical Results

The i-DGFM will be used to solve two arrays and the results thereof will be used to validate the author's implementation.

7.3.1 Applying CEMACS i-DGFM to a square PEC plate array

CEMACS i-DGFM is applied to the one hundred element square plate array at 300 MHz discussed previously and shown in Figure 4.4. The results of the E-field at the red line cut is shown in Figure 7.2 with an error of 6.39%, identical to the MoM, CBFM and DGFM discussed previously. Similar to these methods, the error percentage can be reduced by implementing a near singularity treatment scheme.

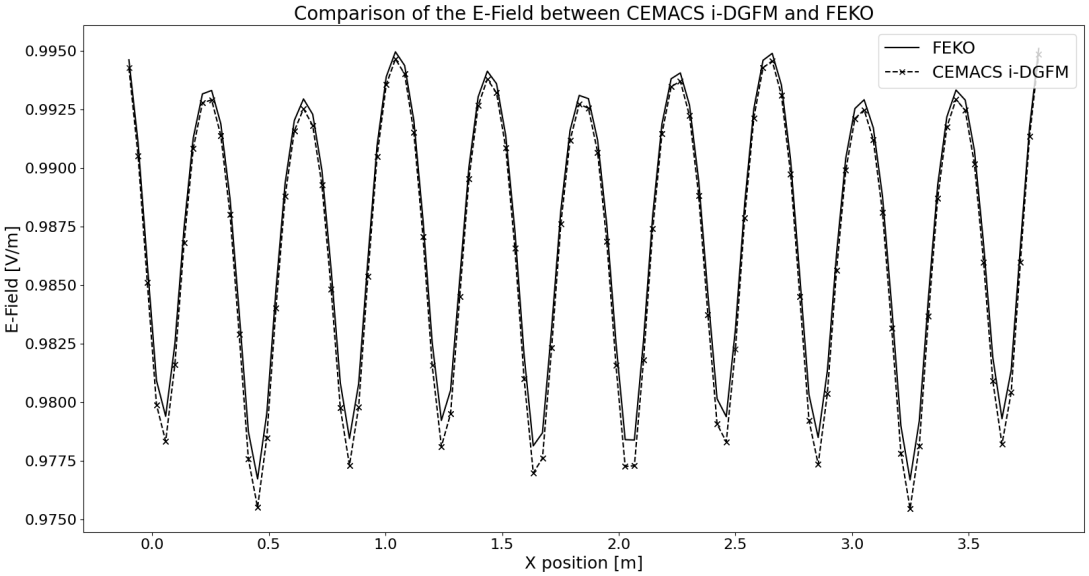


Figure 7.2: A comparison of a square plate array's E-Field between CEMACS i-DGFM and FEKO.

7.3.2 Applying CEMACS i-DGFM to a Vivaldi array

Now, CEMACS i-DGFM will be applied to the Vivaldi array shown in Figure 4.6. The E-plane gain and the magnitude and phase of S_{11} are compared to FEKO and are presented in Figures 7.3, 7.4 and 7.5 respectively. The error percentages for each of these presented in Table 7.1 are sufficiently low. Of note is the error of the E-plane gain which is 9% lower than that of the DGFM, proving the effectiveness of the i-DGFM weight matrix.

7.4 Parallelisation of the i-DGFM

The parallelisation of the i-DGFM is akin to that of the CBFM as many of the calculations are shared i.e the calculation of the primary and secondary basis functions. The differences will be detailed and the process using MPI, a hybrid

MPI-OpenMP approach and CUDA will be discussed below. To obtain the MPI and hybrid MPI-OpenMP results, a single node consisting of 24 cores was used with the specifications found in Appendix A and the results when testing the CUDA GPU implementation were obtained using a Tesla V100 GPU with 5120 CUDA cores (specifications in Appendix B), and an Intel Xeon CPU E5-2690 @ 2.60GHz processor.

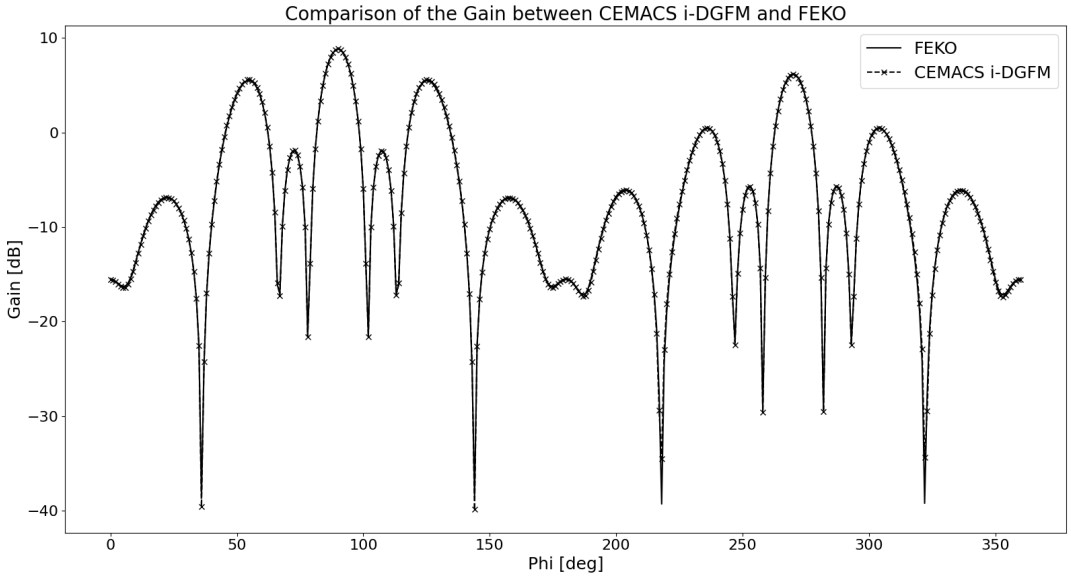


Figure 7.3: A comparison of the E-plane gain (dB) between CEMACS i-DGFM and FEKO for a Vivaldi antenna array.

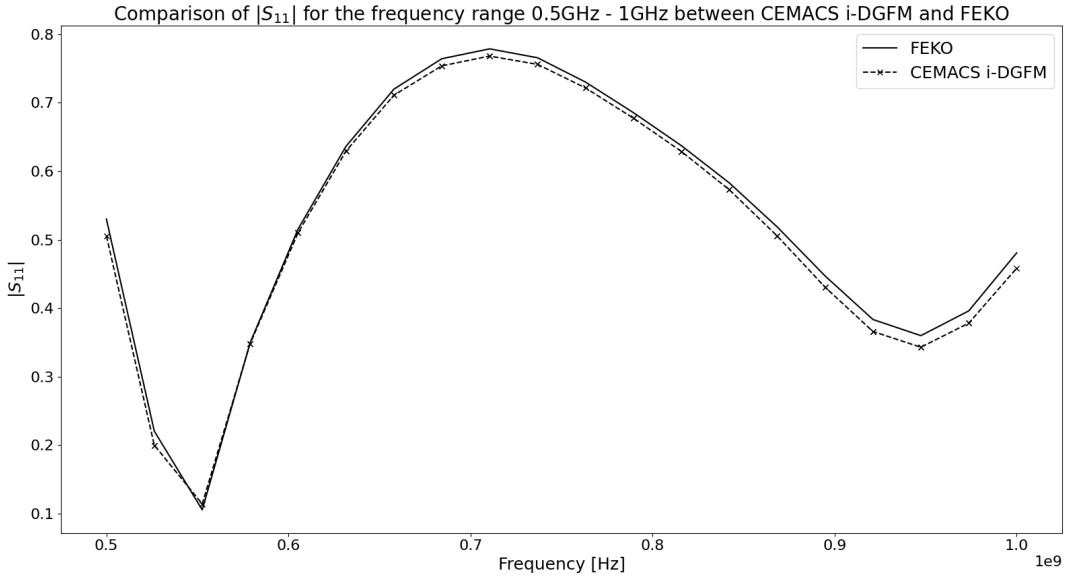


Figure 7.4: A comparison of $|S_{11}|$ between CEMACS i-DGFM and FEKO for a Vivaldi antenna array.

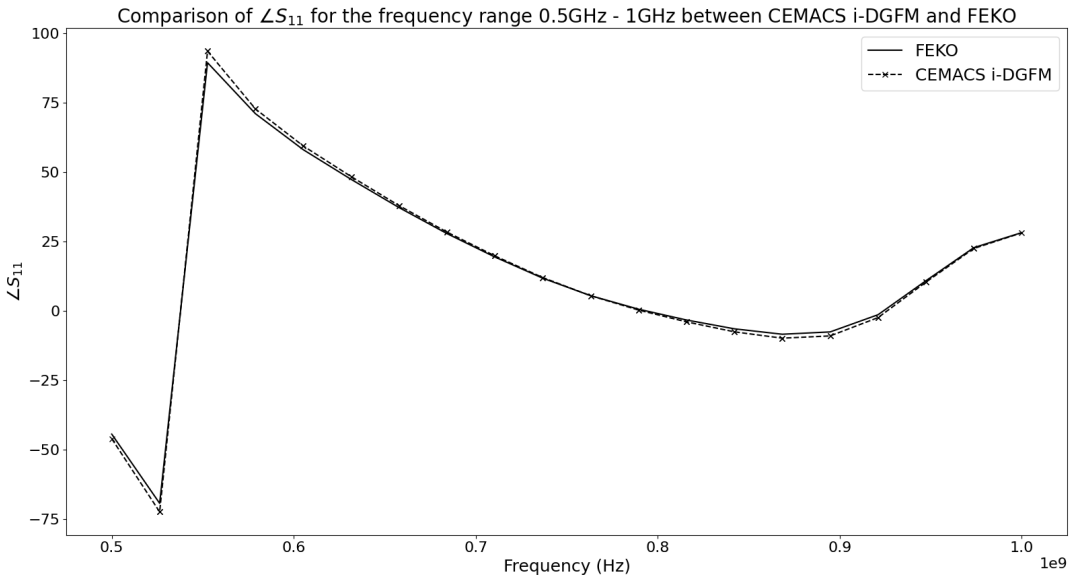


Figure 7.5: A comparison of the phase of S_{11} between CEMACS i-DGFM and FEKO for a Vivaldi antenna array.

Calculated Quantity	$\epsilon_{\%}$
E-plane gain	3.85%
$\lvert S_{11}\rvert$	2.29%
$\angle S_{11}$	3.86%

Table 7.1: Relative error percentages ($\epsilon_{\%}$) between CEMACS i-DGFM and FEKO for three calculated quantities.

7.4.1 MPI

The parallelisation of the i-DGFM using MPI is shown in Figure 7.6 for a simple example consisting of two domains on two processes. The procedure followed is markedly similar to that of the CBFM in Figure 5.7. The difference is that the i-DGFM requires the calculation of i-DGFM weight matrices and the active impedance matrices, $[Z_p^{ACT}]$. Due to this, only $[J_p]$, the primary basis function less the sum of the secondary basis functions on the domain is sent between processes. Compared to the CBFM which requires the transmission of the all basis functions, the transmission requirement is N_i for the i-DGFM versus $N_i \times M$ for the CBFM for each domain.

As seen in Figure 7.7, where the MPI parallelised i-DGFM is applied to 24 element bow-tie array in Figure 5.8, the speed-up increases as the number of MPI processes and number of unknowns increase. This difference in speed-up can be attributed the ratio of MPI communication to work done. Illustrated in Figure 7.8, when the MPI i-DGFM implementation is applied to a square array as in Figure 4.4, the speed-up is not greatly influenced as M changes, which is due to the fact that the same amount of data is being transferred

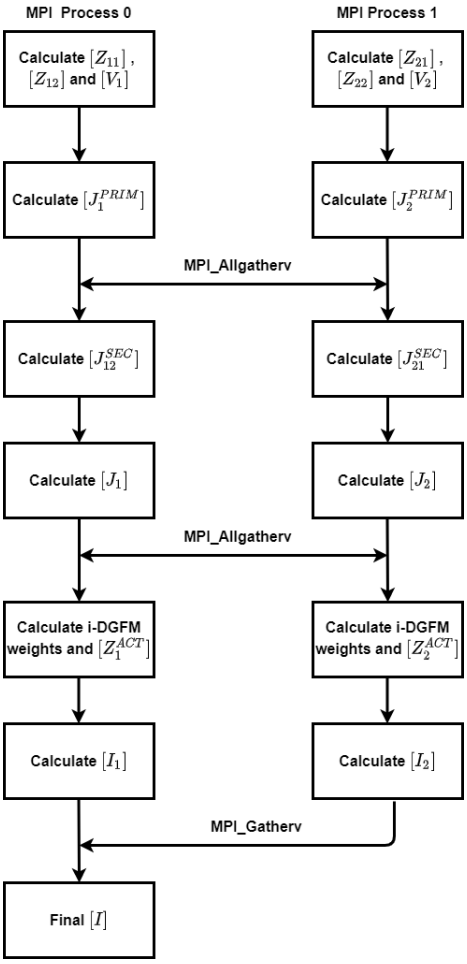


Figure 7.6: Parallelisation of the i-DGFM using 2 domains and 2 MPI processes.

regardless of the value of M. All data is transferred using one dimensional arrays, and if the problem is load balanced, the length of the array should be equal regardless of M.

7.4.2 Hybrid MPI-OpenMP

Considering again the bow-tie array in Figure 5.8, the results of the application of the hybrid MPI-OpenMP technique is shown in Figure 6.10. As in Section 5.4.2, threading was applied to the filling of the blocks of $[Z]$ and $[V]$ as well as the linear algebra routines. Threading is also applied to the calculation to the i-DGFM weights and the resulting calculation of the active impedance matrix. The results in Figure 7.9 follows the overallocation trend of the CBFM (Figure 5.11) discussed in Section 5.4.2. The speed-up decline is however more drastic for i-DGFM in the case where threads are over-allocated (8 processes and 4 threads). While speed-up us satisfactory, this once again proves the importance of understanding the specific hardware used and allocating software

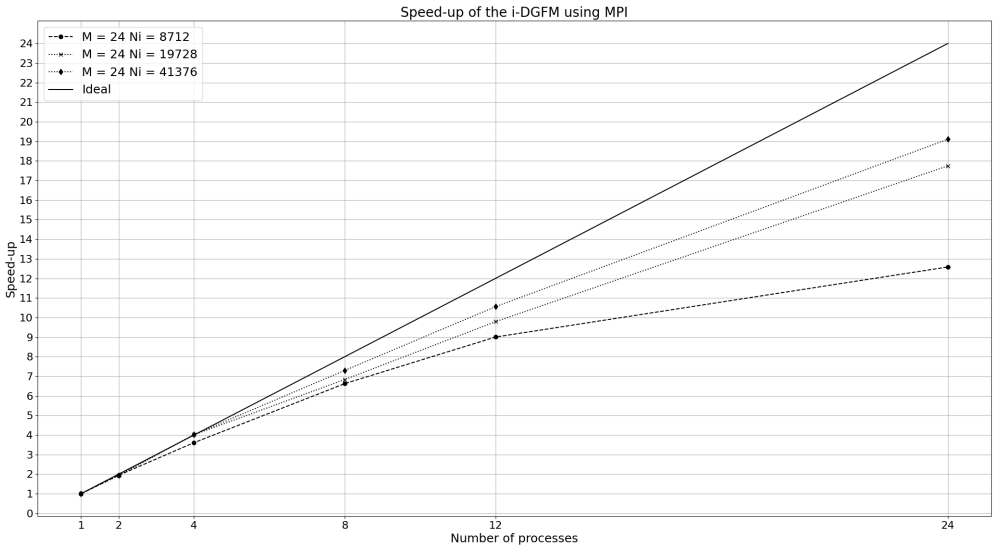


Figure 7.7: Speed-up of the i-DGFM for a bow tie array with variable N

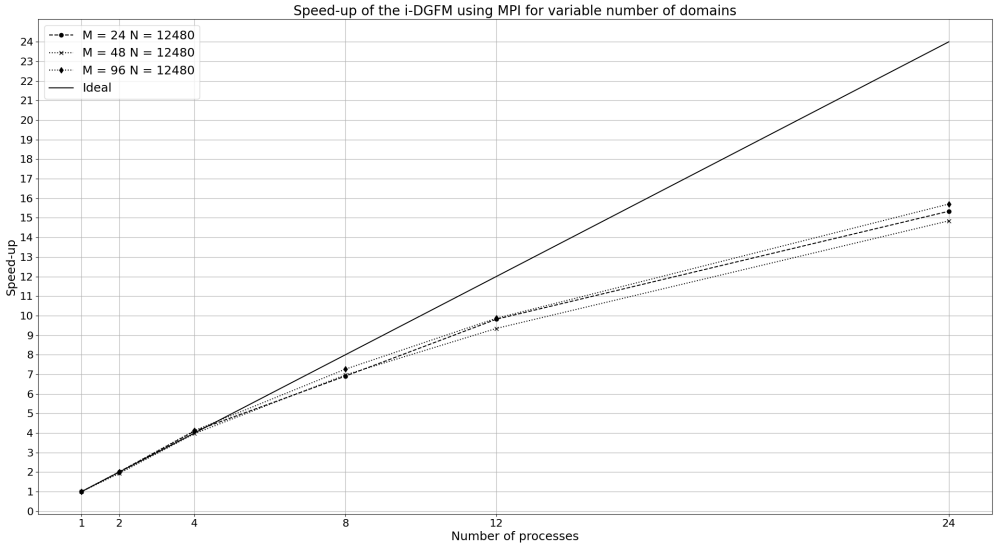


Figure 7.8: Speed-up of the i-DGFM for a square plate array with variable M

resources intelligently.

7.4.3 Implementing the i-DGFM on a GPU using CUDA

The CUDA implementation of the i-DGFM, applied on the twelve element bow-tie array similar to Figure 5.8 on the GPU follows the CBFM where it is faster to keep the entirety of $[Z]$ in the GPU's memory due to the blocks needed for basis function calculation. The speed-up of the i-DGFM when comparing a CUDA implementation on a GPU to a serial i-DGFM implementation on a

CPU is illustrated in Figure 7.10. Comparatively, the speed-up is higher than the CBFM due to the use of fewer linear algebra routines i.e. the i-DGFM weights and subsequent active impedance matrix ($[Z_p^{ACT}]$) were calculated using kernels written by the author. In terms of accuracy, the results obtained are identical to those shown in Section 7.3.

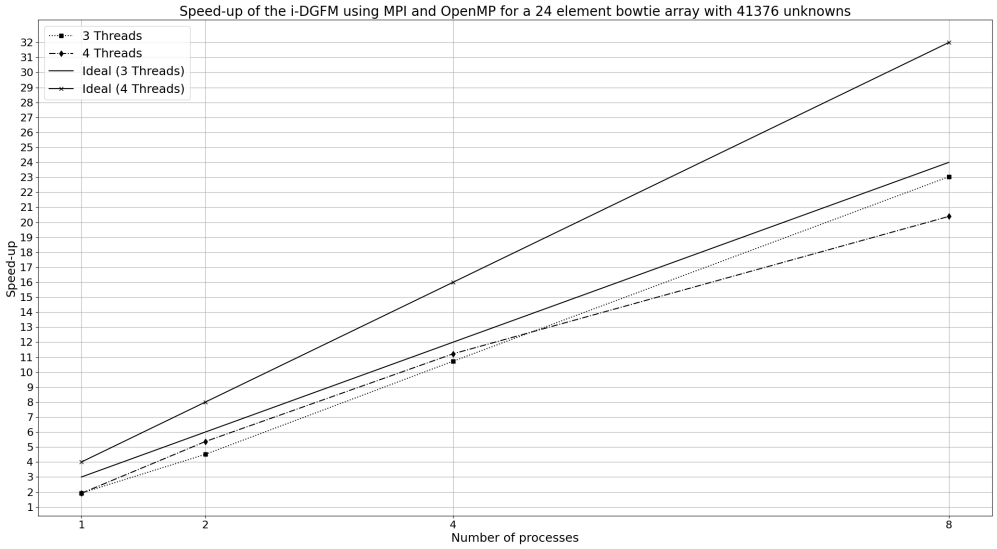


Figure 7.9: Hybrid MPI-OpenMP speed-up using the i-DGFM on a 24 element bow-tie array.

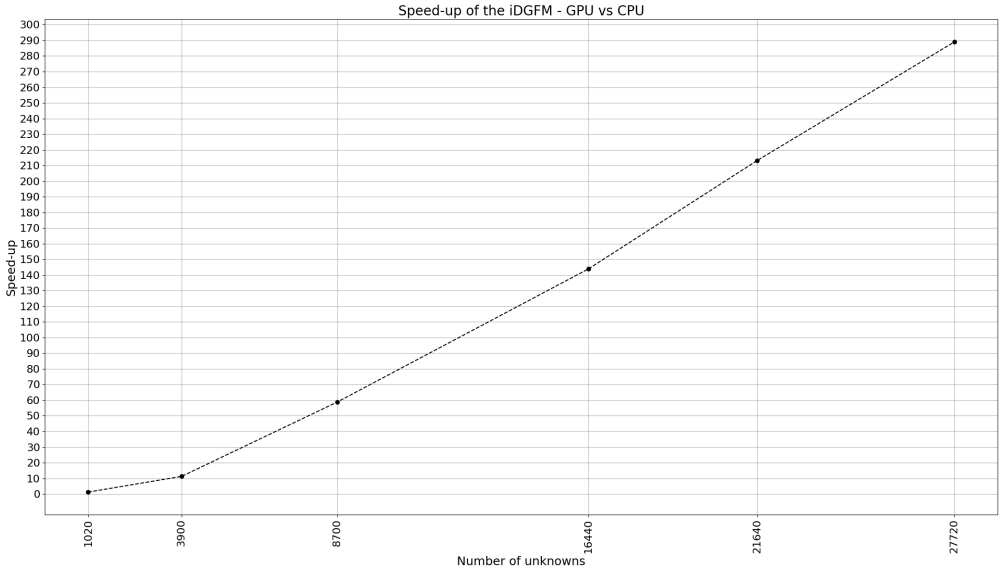


Figure 7.10: The speed-up of the i-DGFM implementation using CUDA on a GPU (Tesla V100) compared to a serial i-DGFM implementation on a CPU(Intel Xeon CPU E5-2690 @ 2.60GHz)

7.5 Conclusion

This chapter presented the formulation of the i-DGFM weight matrix and showed improvement compared to the DGFM in a case of strong mutual coupling. Parallelisation of the i-DGFM was successful both using MPI and OpenMP as well as CUDA on the GPU. In the next chapter a brief comparison will be made regarding the three implemented domain decomposition methods.

Chapter 8

A Comparison of Domain Decomposition Methods

Three domain decomposition techniques, the CBFM (Chapter 5), the DGFM (Chapter 6) and the i-DGFM (Chapter 7) have been investigated, implemented and parallelised. This chapter will briefly condense the differences of the implemented methods as a summary.

8.1 Comparing the implemented domain decomposition methods in terms of memory.

Consider an array where there are M domains consisting of N_i unknowns (RWG basis functions). For an in-core solution i.e. when the complete solution occurs in active memory, both the CBFM and i-DGFM require the complete $[Z]$ in order to calculate basis functions. On the other hand, the DGFM is independent in the sense that only a row of $[Z]$ is required in memory at a time. This memory requirement scales as $\mathcal{O}((M \times N_i)^2)$ for the CBFM and i-DGFM compared to $\mathcal{O}(M \times N_i^2)$ for the DGFM. While the CBFM and i-DGFM require memory for the storage of basis functions ($\mathcal{O}(M^2 \times N_i)$) it is negligible compared to the memory needed by $[Z]$.

For an out-of-core solution, i.e. when $[Z]$ is written to a hard drive and accessed when required, all methods require the same amount of memory which is $\mathcal{O}(M \times N_i^2)$ (a row of $[Z]$). The slow read and write speeds of a hard drive drastically reduce performance when compared to an in-core solution where all data is stored in high speed Random Access Memory (RAM).

Problem	CBFM	DGFM	i-DGFM
$N =$ 8712 using 8 MPI processes	6.9	7.0	6.6
$N =$ 19728 using 8 MPI processes	7.2	7.7	6.8
$N =$ 41376 using 8 MPI processes	8.0	7.5	7.3
$N =$ 8712 using 12 MPI processes	8.9	9.9	9.0
$N =$ 19728 using 12 MPI processes	10.1	10.4	9.8
$N =$ 41376 using 12 MPI processes	11.0	10.4	10.6
$N =$ 8712 using 24 MPI processes	12.4	18.6	12.6
$N =$ 19728 using 24 MPI processes	16.9	19.9	17.8
$N =$ 41376 using 24 MPI processes	18.9	19.9	19.1

Table 8.1: Comparing the MPI speed-up of the CBFM, DGFM and i-DGFM for a 24 element bow tie array with variable number of unknowns (N).

8.2 Comparing the implemented domain decomposition methods in terms of scalability.

The following subsections will compare and discuss the scalability of the CBFM, DGFM and i-DGFM for each of the HPC techniques (MPI, hybrid MPI-OpenMP and CUDA) applied. The results will be tabulated from those presented graphically in Chapters 5, 6 and 7.

8.2.1 MPI

In Tables 8.1 and 8.2 the results from the MPI implementations of the CBFM (Section 5.4.1), DGFM (Section 6.4.1) and i-DGFM (Section 7.4.1) are presented. It is clear that for lower MPI process counts (8 and 12) all domain decompositions perform similarly with the CBFM on average performing the best. This however changes when 24 processes are used, where the DGFM consistently has higher speed-up comparatively. This is expected as the cost of communication of the CBFM and i-DGFM increase with the amount of processes used. The results also clearly show the scalability of all three methods when using a distributed memory HPC technique (MPI).

8.2.2 Hybrid MPI-OpenMP

In Table 8.1, the results from the MPI implementations of the CBFM (Section 5.4.2), DGFM (Section 6.4.2) and i-DGFM (Section 7.4.2) are presented. It is clear from the results that the CBFM benefits the most from a hybrid MPI-OpenMP approach due to the threading of the linear algebra routines used solve the reduced impedance matrix which was solved serially when solely using MPI.

Problem	CBFM	DGFM	i-DGFM
$M = 24$ using 8 MPI processes	7.4	7.1	6.9
$M = 48$ using 8 MPI processes	7.1	7.2	6.9
$M = 96$ using 8 MPI processes	6.9	7.5	7.3
$M = 24$ using 12 MPI processes	10.1	10.0	9.8
$M = 48$ using 12 MPI processes	9.5	9.9	9.3
$M = 96$ using 12 MPI processes	11.0	10.4	10.6
$M = 24$ using 24 MPI processes	9.2	9.9	9.9
$M = 48$ using 24 MPI processes	12.7	17.7	14.8
$M = 96$ using 24 MPI processes	12.2	17.2	15.7

Table 8.2: Comparing the MPI speed-up of the CBFM, DGFM and i-DGFM for a bow tie array with 12480 unknowns and variable number of elements (M).

Problem	CBFM	DGFM	i-DGFM
2 MPI Processes with 3 OpenMP threads	4.8	4.2	4.5
2 MPI Processes with 4 OpenMP threads	5.7	5.2	5.4
4 MPI Processes with 3 OpenMP threads	11.9	9.9	10.7
4 MPI Processes with 4 OpenMP threads	12.7	10.6	11.2
8 MPI Processes with 3 OpenMP threads	23.7	22.3	23.3
2 MPI Processes with 4 OpenMP threads	22.4	21.9	20.4

Table 8.3: Comparing the hybrid MPI-OpenMP speed-up of the CBFM, DGFM and i-DGFM for a 24 element bow tie array with 41376 unknowns.

8.2.3 CUDA on a GPU

In Table 8.1, the results from the MPI implementations of the CBFM (Section 5.4.3), DGFM (Section 6.4.3) and i-DGFM (Section 7.4.3) are presented. While the performance per compute core of the GPU was not tested, it is clear that the DGFM benefits the most from being run using CUDA on a GPU. The low amount of memory available to a GPU (compared to a CPU) will hinder scalability for larger problem sizes. It is therefore necessary to investigate the use of multiple GPUs (a GPU array).

8.3 Comparing the implemented domain decomposition methods in the case of strong mutual coupling.

While the DGFM performs better in terms of memory and scalability, it is weak in accuracy in the case of strong mutual coupling. Earlier in the work, all methods were applied to the same examples and performed well in terms of accuracy except in the case where the DGFM had a much greater error

Problem	CBFM	DGFM	i-DGFM
$N = 1020$	1.4	1.5	1.2
$N = 3900$	13.1	15.8	11.3
$N = 8700$	57.8	64.9	58.8
$N = 16440$	152.9	187.5	143.9
$N = 21640$	202.4	261.4	213.2
$N = 27720$	277.1	345.9	288.9

Table 8.4: Comparing the speed-up of the CUDA implementation on a GPU of the CBFM, DGFM and i-DGFM to their serial implementations on the CPU for a 12 element bow tie array with a variable number of unknowns (N).

Method	E-field $\epsilon_{\%}$	Gain $\epsilon_{\%}$
CBFM	0.02%	0.01%
DGFM	1.72%	1.28%
i-DGFM	0.02%	0.01%

Table 8.5: Comparing $\epsilon_{\%}$ for the CBFM, DGFM and i-DGFM in the case of strong mutual coupling.

percentage, as discussed in Section 6.3.2. This will be investigated further for a case of three bow-tie antennas spaced 5 cm apart at 300 MHz as shown in Figure 8.1. The E-field cut at the red line is shown in Figure 8.2 and a theta cut gain is presented in 8.3. In both figures, the results from the CBFM, DGFM and i-DGFM are compared to CEMACS MoM with the error percentages in Table 8.5. It is clear both visually and from the DGFM's $\epsilon_{\%}$'s that the DGFM performs worse in comparison in the case of strong mutual coupling as stated in Chapter 6.

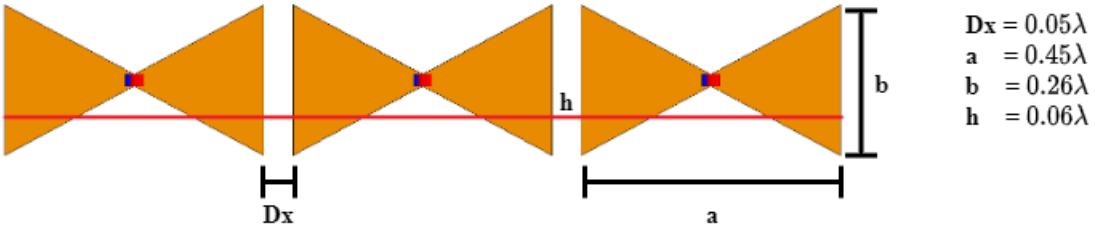


Figure 8.1: Three bow-tie antennas spaced 5cm apart.

8.4 Conclusion

This chapter summarised the differences of the three domain decomposition methods discussed in this work. The key points made were:

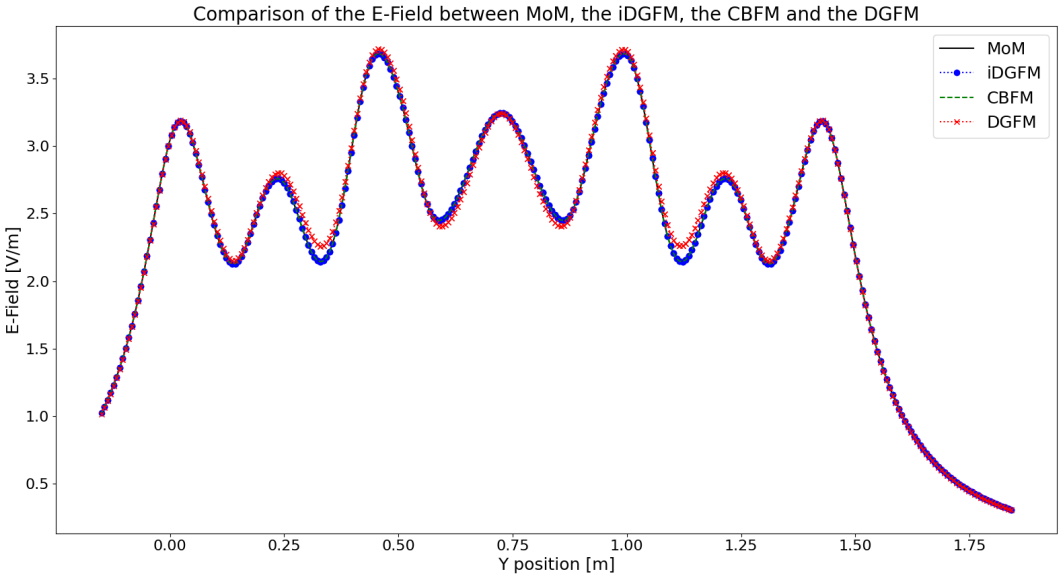


Figure 8.2: A comparison between MoM and the CBFM, DGFM and i-DGFM for the E-field of a three element bow-tie array.

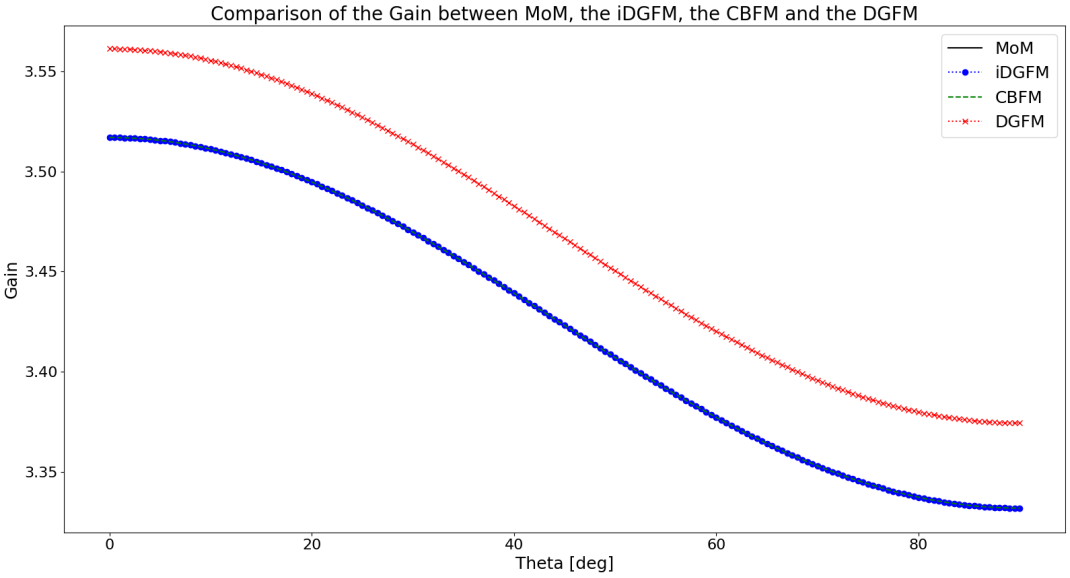


Figure 8.3: A comparison between MoM and the CBFM, DGFM and i-DGFM for the Gain of a three element bow-tie array.

- The CBFM and i-DGFM both require the full $[Z]$ matrix in memory for an in-core solution while the DGFM requires only one row at a time.
- All three methods scale well as problem sizes and computational resources increase. The CBFM scales the lowest due to the creation of a reduced matrix equation compared to the active impedance matrix of the DGFM and i-DGFM.

- For a case of strong mutual coupling, the CBFM and i-DGFM had negligible error compared to the MoM solution while the DGFM is less accurate.

The next chapter will conclude the thesis and present future research aims and improvements.

Chapter 9

Conclusion

In this thesis, three domain decomposition methods were explored, namely the CBFM, DGFM and the i-DGFM. All three were implemented by the author the solution results were positively validated against the commercial CEM software suite, FEKO. High performance computing techniques were investigated and applied successfully, proving the scalability of the three domain decomposition techniques as shown in 8.2. Specifically, MPI and OpenMP were used, as well as CUDA for GPU parallelism. Furthermore, an optimised C++ object orientated framework, CEMACS, was developed. This forms the underlying basis on which the above mentioned domain decomposition solvers are based.

9.1 Thesis review

In Chapter 1, the purpose and background of the research was explained. Chapter 2 then provided insight into high performance computing as well as a critical analysis on the costs and benefits of the chosen domain decomposition techniques. The author's CEM software suite, CEMACS, was then discussed in Chapter 3 with evaluation criteria to discuss the effectiveness of the methods implemented. Chapter 4 discusses the formulation of the MoM and after successful validation provides the foundation for the implementation of the domain decomposition techniques. Chapters 5, 6 and 7 present the formulation, validation and acceleration of the CBFM, DGFM and i-DGFM respectively. The results of all three methods proved positive in both accuracy and acceleration. Finally, a condensed summary was given in Chapter 8 by briefly comparing the three methods.

Per the research objectives, this thesis has demonstrated the successful implementation of three domain decomposition techniques, the CBFM, DGFM and i-DGFM whose accuracy was validated. Further, all three methods were parallelised using MPI, OpenMP and CUDA (for GPU parallelisation) providing positive scaling results.

9.2 Recommendations for future work

1. Implement a near singularity treatment scheme to further improve accuracy.

2. Promising results were found when using a GPU and the use of multiple GPU's should be investigated. This will allow for larger problem sizes as the memory on a single GPU is limited and therefore provide a greater insight into GPU scalability.

3. Implement hybrid domain decomposition solvers, such as the DGFM enhanced CBFM.

4. Implement an algorithm to determine the best domain decomposition solver to use based on the problem. This can be done dynamically at the onset of the solution.

5. Implement a software probe to determine the optimal utilisation of parallel resources for a specific problem (number of threads/number of MPI processes/CUDA grid and block size/etc.)

The points above if implemented, will serve to increase the breadth of CEMACS, the CEM software suite. Implementing Points 1, 2 and 3 will allow the solver to operate on larger problems more accurately. This will also increase the range of solvable problems and produce more data for comparison. Points 4 and 5 are focused more toward a potential end user. If implemented, these would increase the accessibility of CEMACS and allow it to be used more easily.

Appendices

Appendix A

Centre for High Performance Computing (CHPC) Lengau Cluster Node Specifications

A node in the CHPC LENGAU cluster contains a dual socket motherboard and therefore contains two CPUs, namely the Intel Xeon CPU E5-2690 @ 2.60GHz. Each of these CPUs contain 12 cores resulting in 24 cores per node. The detailed specification of a node is

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	24
On-line CPU(s) list:	0-23
Thread(s) per core:	1
Core(s) per socket:	12
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	63
Model name:	Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz
Stepping:	2
CPU MHz:	1286.289
BogoMIPS:	5191.72
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	30720K
NUMA node0 CPU(s):	0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):	1,3,5,7,9,11,13,15,17,19,21,23

Appendix B

Centre for High Performance Computing (CHPC) GPU Specifications

The GPUs available in the CHPC LENGAU cluster are Tesla V100s the detailed specification found below.

Device Name:	Tesla V100-SXM2-16GB
Device Revision Number:	7.0
Global Memory Size:	16914055168
Number of Multiprocessors:	80
Concurrent Copy and Execution:	Yes
Total Constant Memory:	65536
Total Shared Memory per Block:	49152
Registers per Block:	65536
Warp Size:	32
Maximum Threads per Block:	1024
Maximum Block Dimensions:	1024, 1024, 64
Maximum Grid Dimensions:	2147483647 x 65535 x 65535
Maximum Memory Pitch:	2147483647B
Texture Alignment:	512B
Clock Rate:	1530 MHz
Execution Timeout:	No
Integrated Device:	No
Can Map Host Memory:	Yes
Compute Mode:	default
Concurrent Kernels:	Yes
ECC Enabled:	No
Memory Clock Rate:	877 MHz
Memory Bus Width:	4096 bits
L2 Cache Size:	6291456 bytes
Max Threads Per SMP:	2048
Async Engines:	5
Unified Addressing:	Yes
Managed Memory:	Yes
Concurrent Managed Memory:	Yes
Preemption Supported:	Yes
Cooperative Launch:	Yes
Multi-Device:	Yes
Number of CUDA Cores:	5120

List of References

[1] The SKA Project Home Page. Available at: `https://www.skatelescope.org/the-ska-project/`.

[2] Fuller, S.H. and Millet, L.I.: *The Future of Computing Performance: Game Over or Next Level?* 1st edn. The National Academies Press, 2011.

[3] Gregg, C. and Hazelwood, K.: Where is the data? why you cannot debate cpu vs. gpu performance without the answer. pp. 134–144. 04 2011.

[4] Rao, S., Wilton, D. and Glisson, A.: Electromagnetic scattering by surfaces of arbitrary shape. *IEEE Transactions on Antennas and Propagation*, vol. 30, no. 3, pp. 409–418, 1982.

[5] Arecibo Observatory Home Page. Available at: `http://www.naic.edu/`.

[6] Ebrahim, T. and Ludick, D.J.: A parallelized fast array analysis approach. In: *2020 14th European Conference on Antennas and Propagation (EuCAP)*, pp. 1–3. 2020.

[7] Davidson, D.B.: *Computational Electromagnetics for RF and Microwave Engineering.* 2nd edn. Cambridge, 2011.

[8] Jin, J.: *The Finite Element Method in Electromagnetics.* Wiley - IEEE. Wiley, 2015. ISBN 9781118842027.
Available at: `https://books.google.co.za/books?id=DFi-BgAAQBAJ`

[9] Kunz, K. and Luebbers, R.: *The Finite Difference Time Domain Method for Electromagnetics.* Taylor & Francis, 1993. ISBN 9780849386572.
Available at: `https://books.google.co.za/books?id=00on9fRvJqIC`

[10] Prakash, V. and Mittra, R.: Characteristic basis function method: A new technique for efficient solution of method of moments matrix equations. *Microwave and Optical Technology Letters*, vol. 36, pp. 95 – 100, 2003.

[11] Ludick, D.J., Maaskant, R., Davidson, D.B., Jakobus, U., Mittra, R. and de Villiers, D.: Efficient analysis of large aperiodic antenna arrays using the domain green's function method. *IEEE Transactions on Antennas and Propagation*, vol. 62, no. 4, pp. 1579–1588, 2014.

[12] Ludick, D.J.: *Efficient numerical analysis of finite antenna arrays using domain decomposition methods.* Ph.D. thesis, Stellenbosch University, 2014.

[13] MPICH Home Page. Available at: `https://www.mpich.org/`.

[14] OpenMP Home Page. Available at: `https://www.openmp.org/`.

[15] Intel MPI Home Page. Available at: `https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html`.

[16] Vinter, B., BJØRNDALEN, J., ANSHUS, O. and LARSEN, T.: A comparison of three mpi implementations. vol. 62, 09 2010.

[17] OpenCL Home Page. Available at: `https://www.khronos.org/opencl/`, .

[18] CUDA Home Page. Available at: `https://developer.nvidia.com/cuda-toolkit`.

[19] OpenACC Home Page. Available at: `https://www.openacc.org/`, .

[20] CHPC Home Page. Available at: `https://www.chpc.ac.za/`.

[21] Altair FEKO Home Page. Available at: `https://altairhyperworks.com/product/FEKO`.

[22] LAPACK Home Page. Available at: `http://www.netlib.org/lapack/`.

[23] OpenBLAS Home Page. Available at: `https://www.openblas.net/`, .

[24] cuBLAS Home Page. Available at: `https://developer.nvidia.com/cublas`.

[25] cuSOLVER Home Page. Available at: `https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html`.

[26] Khayat, M.A. and Wilton, D.R.: Numerical evaluation of singular and near-singular potential integrals. *IEEE Transactions on Antennas and Propagation*, vol. 53, no. 10, pp. 3180–3190, 2005.

[27] Li, L., Wang, K. and Eibert, T.: A new radial-angular-r2 transformation for singular integrals on triangular meshes. pp. 565–568. 04 2014.

[28] Makarov, S.: Mom antenna simulations with matlab: Rwg basis functions. *IEEE Transactions on Antennas and Propagations*, vol. 43, pp. 100–107, 2001.

[29] Suter, E. and Mosig, J.R.: A subdomain multilevel approach for the efficient mom analysis of large planar antennas. *Microwave and Optical Technology Letters*, vol. 26, no. 4, pp. 270–277, 2000.

[30] Craeye, C., Laviada, J., Maaskant, R. and Mittra, R.: Macro basis function framework for solving maxwell's equations in surface integral equation form. 2014.